Evaluating The Vulnerability Detection Efficacy Of Smart Contracts Analysis Tools

Silvia Bonomi[1], Stefano Cappai[1], and Emilio Coppa[2]

[1] Sapienza University of Rome, Rome, Italy
[2] LUISS University, Rome, Italy

Category	Vulnerability name	
Blockchain	B1	Frontrunning
	B2	Erroneous Blockchain Assumption
	B3	Bad Randomness
EVM	E1	Gas Limit DoS
Business Logic	L0	Error in Business Logic Model
	L1	Freezing Active Position
	L2	Erroneous Accounting Business Implementation
	L3	Missing Business Logic Checks
	L4	Wrong Implementation
Solidity	S1	Reentrancy
	S2	Precision Loss
	S3	Integer Overflow and Underflow
	S4	Using `tx.origin`
	S5	Arbitrarily Code Injection
	S6	Outdated Compiler Bug
	S7	Unsafe Casting
	S8	Using `transferFrom()` instead of `safeTransferFrom()` in ERC721 implementation
	S9	Using `_mint()` instead of `_safeMint()` in ERC721 implementation
	S10	Dangerous `DelegateCall`
	S11	Relying on External Source
	S12	Division by Zero DoS
	S13	Temporal DoS
	S14	Erroneous ERC721 Implementation
	S15	Erroneous `Pausable` implementation

Table 1: Overview on well-known smart contract issues.

1 A list of vulnerabilities of smart contracts

In this technical report, we describe a list of well-known vulnerabilities of smart contracts and analyze the vector attacks using examples of smart contracts inspired by real-world exploits, Capture-the-Flag competitions, or written by us.

In Table 1 there is the list of issues we analyze in this section. Every issue description starts with a table that resumes the vulnerability and the corresponding *CWE*, *SWC-ID*, and *Demystifying Paper* references, if any. Then we report a discussion about our scoring decisions and an overview of the vulnerability. Note: we decided to start with *L0* in *Business Logic* group because we consider *L0 - Error in Business Logic Model* a high-level problem, related to the business model and not to the code. We analyze it in Section 1.5.

1.1 B1 - Frontrunning

Blockchain	**B1**	**Frontrunning**
References	CWE	362- Concurrent Execution using Shared Resource with Improper Synchronization ('Race Condition')
	SWC-ID	114 - Transaction Order Dependence
	Demystifying	S1.2 - Sandwich attack

Vulnerability description This vulnerability takes its denomination from an illegal practice in standard finance trading. Let's think about an investor Bob who asks a dealer to buy a huge amount of an asset on the market. The dealer Eve could maliciously buy a certain quantity of the same asset before placing an investor order. In this way, knowing that the price of the asset will increase after the investor purchases, Eve will have a sure profit selling her asset at the new price. The same concept can be applied to the blockchain environment. In general, the validator node builds a new block using transactions. Usually, it prioritizes transactions with respect the gas fees. So, transactions with higher gas fees will be executed first than others. Furthermore, there is a time frame when the nodes collect transactions and put them into their local *mempools* before building the block. Every node has a *mempool*, that could slightly differ from another's one. When a node receives a transaction, it propagates that transaction over the network. A malicious actor that can read a node *mempool*, could be able to foresee transactions that make profits and could propose their version of the same transaction, with higher gas fees. This transaction will be executed before the victim's. In an advanced application of this exploit, called a *sandwich attack*, an attacker could create two transactions, one before (*frontrunning*) and one after (*backrunning*) the victim transaction.

An important concept related to Frontrunning is *MEV (Maximal Extractable Value)*. We report *MEV* definition from the Ethereum website[3]: "the maximum value that can be extracted from block production in excess of the standard block reward and gas fees by including, excluding, and changing the order of transactions in a block". So, we have to distinguish between a good MEV and a bad MEV, obtained by malicious blockchain users.

We want to underline that this attack can be well exploited by nodes because they have transaction information before anything else. However, everybody can subscribe to apps (e.g., using Infura) that supply WebSocket services where pending transactions are continuously pushed.

[3] `https://ethereum.org/en/developers/docs/mev/`

Now, we want to describe a simple example of Frontrunning taken from *Solidity by Example* website[4] and slightly modified:

```
 1 // SPDX-License-Identifier: MIT
 2 pragma solidity ^0.8.0;
 3
 4 contract FindThisHash {
 5     bytes32 public constant hash =
 6 0
       x564ccaf7594d66b1eaaea24fe01f0585bf52ee70852af4eac0cc4b04711cd0e2
       ;
 7
 8     bool alreadyClaimed = false;
 9
10     constructor() payable {}
11
12     function solve(string memory solution) public {
13         require(!alreadyClaimed);
14         require(hash == keccak256(abi.encodePacked(solution))
               , "Incorrect answer");
15
16         alreadyClaimed = true;
17         (bool sent, ) = msg.sender.call{value: 10 ether}("");
18         require(sent, "Failed to send Ether");
19     }
20 }
```

This contract rewards who guess `solution` string, i.e., who guess what string, after *sha256* hashing, is equal to `hash` value (line 14). After hundreds of hours of effort, Bob finally managed to find `solution` string and send his transaction to `FindThisHash` contract, setting the gas price to 1 gwei: when a new block will be mined, he will gain 10 ETH. Eve is listening on *mempool* and finds Bob's transaction. She understands that she could replicate Bob's transaction and put it before, setting the gas price to 10 gwei. In this way, Eve's transaction will be processed before then Bob's one. `FindThisHad` contract sends 10 ether to Eve and Bob's transaction reverts, because `alreadyClaimed` variable is `true` after Eve's transaction.

According to [1] the attacker can exploit frontrunning mainly in three ways: the *displacement attack*, where the hacker inserts its/her transaction before the victim one, the *insertion attack* or *sandwich attack*, where the hacker succeeds to insert victim's transaction between two malicious transactions, and the *suppression attack*, where the attacker exploit the volume of transactions on the network and manage the victim's transaction to be inserted in the current block.

In the last years, DApps have implemented some mitigation for this issue. The most common is to introduce some slippage restrictions that manage to contain the *frontrunners* malicious profits. However, there is no standard mitigation approach, and several studies worked on it [2][3]. Frontrunning appears to be

[4] `https://solidity-by-example.org/hacks/front-running/`

one of the most prominent exploits in DeFi security. It is estimated that to date about $280 million are drained by *frontrunners* every month [4]. In [5], the authors estimated a loss of $174.34 million due to frontrunning in the period between 1st December 2018 and the 5th August 2021.

1.2 B2 - Erroneous Blockchain Assumption

Blockchain	**B2**	**Erroneous Blockchain Assumption**
References	CWE	829 - Inclusion of Functionality from Untrusted Control Sphere
	SWC-ID	116 - Block values as a proxy for time
	Demystifying	-

Vulnerability description This vulnerability is caused by erroneous assumptions on the blockchain environment. For example, a developer could erroneously assume that on Ethereum the number of seconds per block is 12, while it can change over time due to traffic on the network. As a best practice, smart contracts should be built agnosticically concerning the underlying blockchain. Otherwise, a future blockchain update could break the contract's logic. Moreover, several DApps are built to be executed on more than one blockchain (Ethereum, Optimism, Arbitrum, etc.), and too strong assumptions on blockchain behavior could lead to different contract logic on different blockchains.

1.3 B3 - Bad Randomness

Blockchain	**B3**	**Bad Randomness**
References	CWE	330 - Use of Insufficiently Random Values
	SWC-ID	120 - Weak Sources of Randomness from Chain Attributes
	Demystifying	

Vulnerability description Transactions on blockchain are deterministic. The code inside smart contracts has no source of randomness. This means that every attempt to generate random things with smart contracts is doomed to fail. In the last years, developers have been able to work around this problem using oracles, which can represent a trusted source of randomness. To explain this issue, we report an example from an Ethernaut contest[5]:

```
// SPDX-License-Identifier: MIT
pragma solidity ^0.8.0;

contract CoinFlip {

  uint256 public consecutiveWins;
  uint256 lastHash;
  uint256 FACTOR = 57896044618658097711785492504343953926634992
  2332820282019728792003956564819968;

  constructor() {
    consecutiveWins = 0;
  }

  function flip(bool _guess) public returns (bool) {
    uint256 blockValue = uint256(blockhash(block.number - 1));

    if (lastHash == blockValue) {
      revert();
    }

    lastHash = blockValue;
    uint256 coinFlip = blockValue / FACTOR;
    bool side = coinFlip == 1 ? true : false;

```

[5] https://ethernaut.openzeppelin.com/level/3

```
26     if (side == _guess) {
27       consecutiveWins++;
28       return true;
29     } else {
30       consecutiveWins = 0;
31       return false;
32     }
33   }
34 }
```

In this contract, there is a `flip()` function that tries to replicate a random behavior. It saves into `blockValue` variable the hash of `block.number-1` (line 16). We remind that `block.number` is a *Special Variable* which returns the value of the current block. After that, `flip()` function computes the value of `coinFlip`, dividing `blockValue` by `FACTOR` value (line 23). Finally, the function computes the boolean `side` variable, which represents the heads or tails result of a flipped coin. An attacker could replicate this function on the same block and forecast the result of flipping coins. Indeed, the `side` value depends only on `block.number`, which can be foreseen.

1.4 E1 - Gas Limit DoS

EVM	**E1**	**Gas Limit DoS**
References	CWE	400 - Uncontrolled Resource Consumption
	SWC-ID	128 - DoS With Block Gas Limit
	Demystifying	L4 - Bugs that are caused by exceeding the gas limitation

Vulnerability description This vulnerability is based on gas fees and in particular on *block gas limit*. If a contract is vulnerable, an attacker can arbitrarily increase the gas cost of a subset of the contract's functionalities. In this way, other users will not be able to use those functionalities anymore. Let's see an example written by us:

```solidity
// SPDX-License-Identifier: MIT
pragma solidity ^0.8.0;

contract GasLimitDoS{

    // An address is associated with the fund. This is a "position"
    mapping(address => uint256) public position;

    uint256 loops = 1;
    uint256 public expensive_storage_variable = 0;

    /*  deposit function can be used to send funds to this contract.
    */
    function deposit() public payable {
        position[msg.sender] += msg.value;
    }

    /*  withdraw function can be used to withdraw a funds amount from this contract.
        @parameter amount: the amount a user would to withdraw

        At the beginning of the function, we have inserted a loop of an
        expensive function that an attacker can exploit.
    */
    function withdraw(uint256 amount) public{
        for(uint256 i=0;i<loops;i++){
```

```
26                 _gasExpensiveFunction();
27             }
28             require(position[msg.sender] >= amount);
29             position[msg.sender] -= amount;
30             payable(msg.sender).transfer(amount);
31         }
32 
33         function _gasExpensiveFunction() internal{
34             // a gas expensive function
35         }
36 
37         // A function that an attacker can call to arbitrarily
               modify the gas cost of withdrawal operation
38         function changeLoops(uint256 newLoopsValue) public{
39             loops = newLoopsValue;
40         }
41     }
```

`GasLimitDoS` contract has a function, `changeLoops()`, which permit to change the value of state variable `loops` arbitrarily (line 39). Indeed, `changeLoops()` is defined as `public`, i.e., it can be called by any other smart contract or EOA. However, a loop inside `withdraw()` function depends on `loop` variable (lines 25-29). So, an attacker can force the loop inside `withdraw(uint256 amount)` to iterate any number of times (at most $2^{255} - 1$). Inside this loop, there is `_gasExpensiveFunction()`: it is a function that has high gas cost. According to network conditions and the current gas block limit, the attacker can set the `loops` variable to force `withdraw()` function to revert for everyone would call it. In other words, the attacker can force every transaction that involves `withdraw()` function to overcome the block limit (whatever it is). An attacker could make `withdraw(uint256 amount)` so expensive that nobody will able to call it.

In [6] authors describe several DoS attacks on smart contracts. Beyond the *Gas Limit DoS* attack, they describe the so-called *Gas Limit DoS on the Network via Block Stuffing*. This exploit is a bit different from the above one: the attacker, instead of force reverting of the transaction, prevents every other transaction than his/her from being included in new blocks. To do so, he/she has to create several transactions, that can be very expensive. However, in some cases, the final award can compensate for the effort. For example, this attack was successfully exploited on Fomo3D [7].

1.5 L0 - Error in Business Logic Model

Business Logic	**L0**	**Error in Business Logic Model**
References	CWE	-
	SWC-ID	-
	Demystifying	SC - Contract implementation-specific bugs

Vulnerability description This problem is due to an *erroneous model of business logic*. This kind of issue is not related to the smart contract implementation. Even if the implementation is correct, the business model on which it relies could have some leaks. These leaks could be not a problem in a traditional environment but can lead to unexpected behaviors in a distributed one. Sometimes, there could be choices in developing DApp that could have some side effects. For example, an erroneous rewards logic could lead users to quit, reducing decentralization.

This kind of problem belongs to a very high level (so we use L0 ID). We want to underline that this issue is strictly related to business logic and many times not even a tool that knows the business model could be able to detect because even the model's invariants could be not correct. This kind of issue can be found in having a lot of experience with distributed systems, smart contracts, and, sometimes, game theory [8].

1.6 L1 - Freezing Active Position

Business Logic	**L1**	**Freezing Active Position**
References	CWE	-
	SWC-ID	-
	Demystifying	S3.2 - Incorrect state updates

Vulnerability description This issue mainly appears in lending protocols and AMMs in general. It consists of an erroneous implementation of business logic that causes a user's position (i.e. the user state in a contract that allows him to deposit and withdraw funds) to be closed when there are still funds in it.

We have to distinguish this vulnerability from the intended act of closing a user's position by developers. When this vulnerability occurs, nobody can access frozen funds anymore, not even the contract owners. Let's see an example written by us:

```solidity
// SPDX-License-Identifier: MIT
pragma solidity ^0.8.0;

contract FreezingFunds{

    // Funds are split into two different buckets: fund1 and
       fund2
    struct Funds {
        uint256 fund1;
        uint256 fund2;
    }

    // An address is associated with the two funds. This
       defines a "position"
    mapping(address => Funds) public position;

    /*  deposit function can be used to send funds to this
       contract.
        @parameter fund_index:
            0: funds are sent to fund1
            1: funds are sent to fund2
    */
    function deposit(bool fund_index) public payable {
        if(fund_index == true) position[msg.sender].fund1 +=
            msg.value;
        else position[msg.sender].fund2 += msg.value;
    }

25      /*  withdraw function can be used to withdraw a funds
            amount from this contract.
26          @parameter fund_index:
27              0: funds are withdrawn from fund1
28              1: funds are withdrawn from fund2
29          @parameter amount: the amount a user would to
                withdraw
30
31          When a user fund is empty, this function closes the
                user's position
32      */
33      function withdraw(bool fund_index, uint256 amount) public
            {
34          if(fund_index == true){
35              require(position[msg.sender].fund1 >= amount);
36              position[msg.sender].fund1 -= amount;
37              payable(msg.sender).transfer(amount);
38
39              // close position
40              if(position[msg.sender].fund1 == 0) delete(
                    position[msg.sender]);
41          }
42          else{
43              require(position[msg.sender].fund2 >= amount);
44              position[msg.sender].fund2 -= amount;
45              payable(msg.sender).transfer(amount);
46
47              // close position
48              if(position[msg.sender].fund2 == 0) delete(
                    position[msg.sender]);
49          }
50
51      }
52  }
```

In this case, the contract can assume an unwanted behavior and this could lead to damages to developers and users. Initially, Bob deposits 1 wei in `fund1` and 1 wei in `fund2` using `deposit()` function twice. This function is `payable`, so it accepts calls with ethers. `deposit()` function add automatically ethers to the `FreezingFunds` balance and update the `position` values (lines 20-23). After the deposit operation, Bob decides to withdraw 1 wei from `fund1`, using the `withdraw()` function. When he does that, the `FreezingFunds` contract closes the user's position: in line 40, the `withdraw()` function `delete` the `position` value corresponding to `msg.sender` (i.e., the address of Bob). Closing the user's position, the contract also deletes `position[msg.sender].fund2`. After that, Bob has no way to retrieve funds from `fund2`. Moreover, nobody else can recover those funds, even if they belong to `FreezingFunds` balance, because this smart contract has no other functions than the `withdraw()` one to retrieve funds.

1.7 L2 - Erroneous Accounting Business Implementation

Business Logic	**L2**	**Erroneous Accounting Business Implementation**
References	CWE	670 - Always-Incorrect Control Flow Implementation
	SWC-ID	-
	Demystifying	S6 - Erroneous accounting

Vulnerability description This issue is quite general and covers several kinds of bugs and vulnerabilities. Usually, erroneous accounting can't be exploited, but users' interactions with the contract can change its state in an inconsistent one, according to business logic. For example, an erroneous calculating order could break some business invariant that developers fixed in that contract.

In some cases, this issue could lead to a wrong semantic of an important variable (e.g. an erroneous accounting of the smart contract's balance variable could lead to a different value between that variable and the true balance). We wrote an example inspired by the governance mechanism developed in many DApps. To make the exposition clear, we have to introduce some concepts:

– ***Treasury***: it represents the amount of wei inside the contract. It must be equal to the contract's balance;
– ***Proposal***: *proposals* can be sent by everybody. It is represented by a string, and saved into `proposals` string array. When there are 10 proposals inside the contract, it starts the voting period;
– ***Voting period***: during this period, voters can vote for 1 of the 10 proposals in `proposals`. Their votes are saved in `votes`. Furthermore, the contract uses `canVote` mapping to avoid multiple votes by the same voter.

There are three roles:

– ***owner***: this address builds the contract and puts the initial treasury inside. Only *owner* can call `endVoteAndRewards()`;
– ***voters***: these addresses can vote among *proposals*.
– ***proposers***: everyone can be a proposer, even the *owner*, and *voters*. A *proposer* adds a new *proposal* using `addProposal()`. Proposers can add only one proposal per *voting period*.

When a *voting period* ends, the contract checks what *proposal* has more votes. That *proposal* will have a reward of 5% of the treasury. Then, the contract rewards also the *voters*, as an incentive for their participation. This reward is 3% of the *treasury*, split among *voters*. We want to describe the `ErroneousAccounting` contract by splitting it into several parts.

Initial settings and the `constructor()` In these lines we define the `contract`, the state variables and their initial values, and the `constructor()` function. `ErronousAccounting contract` doesn't inherit from other objects and doesn't define `Using` statements. It has several state variables. In `uint256 treasury` there is the amount of wei in the smart contract; it should be equal to the smart contract's balance. The `owner` is the address of the *owner*, the special role defined above. `inVotingPeriod` is a boolean variable which represents if the *voting period* is active. Then we define two arrays, `voters` and `proposal`, where the addresses of *voters* and the strings that represent *proposals* are stored. In line 21 we define the `votes mapping`, i.e., correspondence between every *proposal* and its number of *votes*, Finally, we define `canVote` mapping to track addresses that have already voted and `choosenProposals` to save the winner proposals and their corresponding rewards.

```
1  // SPDX-License-Identifier: MIT
2  pragma solidity ^0.8.0;
3
4  contract ErroneousAccounting{
5
6      uint256 public treasury = 0;
7      address owner;
8
9      //If inVotingPeriod is true, the contract is inside the
           voting period
10     bool public inVotingPeriod = false;
11
12     address[] voters = [
13        // list of addresses that can vote
14     ];
15
16     // List of proposals. It should contain at most 10
           proposals.
17     string[] proposals;
18
19     // A mapping between the proposal index and the number of
            votes.
20     // This mapping tracks the number of votes for a proposal
            index.
21     mapping(uint8 => uint256) votes;
22
23     // A mapping between voter address and a boolean.
24     // If canVote is false, that address has already voted.
25     mapping(address => bool) canVote;
26
27     // The list of chosen proposals after the voting period.
28     mapping(uint256 => uint256) choosenProposals;
29
30     // Constructor sends initial treasury and sets the owner
           of the contract
```

```
31      constructor() payable{
32          treasury = msg.value;
33          owner = msg.sender;
34      }
```

Adding and voting proposal mechanism In this paragraph we describe two methods of `ErroneousAccounting contract`: `addProposal` and `voteProposal`. Both functions have `public` visibility, i.e., can be called by every other smart contract and/or EOA.

In lines 43 and 44, the former method checks if the `inVotingPeriod` is `false`, i.e., if the *voting period* is active and *proposals* can be suggested. Then, it adds `_description` string to the `proposals` array: it can be voted in the next voting phase. Finally, the function calls the `_startVotingPeriod` function and starts the *voting period* if there are 10 *proposals* into the `proposals` array.

The `voteProposal` method, instead, permits *voters* to express their preferences among *proposals* into the `proposals` array. It makes some initial checks. It verifies that the selected *proposal* exists; then it checks that the *voting period* is started and the *voter* hasn't voted yet. If these conditions are true, `voteProposal` function adds the vote to `idProposal` inside the `votes mapping` and sets `canVote` value of the *voter* address to `false`.

```
35
36      /*
37      This method permits everybody to add a new proposal, out
            from the voting period.
38      The same address can add more than one proposal.
39      If the added proposal is the tenth, this method starts
            the voting period.
40      @param _description: a string representing proposal.
41      */
42      function addProposal(string memory _description) public{
43          require(!inVotingPeriod,
44              "No proposal accepted inside voting period");
45          proposals.push(_description);
46          if(proposals.length == 10){
47              _startVotingPeriod();
48          }
49      }
50
51      /*
52      Only voters can call this method and only during the
            voting period.
53      A voter can vote once.
54      @param idProposal: a number between 0 and 10 that
            represents the proposal that the voter wants to
            select.
55      */
56      function voteProposal(uint8 idProposal) public {
```

```
57         require(idProposal <= proposals.length - 1,
58             "Index of proposal is not valid");
59         require(inVotingPeriod, "Out from voting period");
60         require(canVote[msg.sender],
61             "This address has already voted");
62         votes[idProposal]++;
63         canVote[msg.sender] = false;
64     }
```

The distribution of rewards among the winner proposal and voters at the end of voting period In this paragraph we describe the `endVoteAndRewards()`. It is a `public` function but can be called successfully only by the *owner* address, thanks to `require` check in line 75. The method iterates over the `votes` array and figures out what *proposal* obtained more *votes*, saving the corresponding string description into the `chosenProposal` variable and the number of obtained votes into the `chosenProposalVotes` variable. Then, in lines 86-87, the function computes the `proposalReward` and the `voterRewards`. In line 89, it saves into `chosenProposals` the winner *proposal* and corresponding reward, using the hash of *proposal* string description as the index. Then, in lines 90-92 `endVoteAndRewards` transfers the rewards to *voters* addresses. At the end, it decreases `treasury` variable by the `proposalReward` (but not also by the `voterReward` amounts) and calls the `_endVotingPeriod()` method.

```
65
66     /*
67     Only the owner can call this function and finish the
           voting period.
68     This method finds the proposal that received more votes
           and saves it
69         into the choosenProposals mapping.
70     Then, it assigns 5% of the treasury to the winner's
           proposal and 3% of the treasury among
71         all voters.
72     Finally, it decreases the amount of treasury and ends the
           voting period
73     */
74     function endVoteAndRewards() public{
75         require(msg.sender == owner, "Only owner can finish
               voting period");
76         string memory chosenProposal;
77         uint256 chosenProposalVotes = 0;
78         for(uint8 i; i<10;i++){
79             if(votes[i] == 0) continue;
80             else if(votes[i] > chosenProposalVotes){
81                 chosenProposal = proposals[i];
82                 chosenProposalVotes = votes[i];
83             }
84         }
85
86          uint256 proposalReward = (treasury*5)/100;
87          uint256 voterReward = ((treasury*3)/100)/voters.
                length;
88
89          choosenProposals[uint256(keccak256(abi.encodePacked(
                chosenProposal)))] += proposalReward;
90          for(uint256 i; i< voters.length; i++){
91              payable(voters[i]).transfer(voterReward);
92          }
93
94          treasury -= proposalReward;
95
96          _endVotingPeriod();
97      }
```

Internal utility functions The _startVotingPeriod and _endVotingPeriod functions are utility internal methods, i.e., they can be called only by other methods inside the ErroneousAccounting contract. The former starts the *voting period*, setting the inVotingPeriod boolean variable and canVote mapping to true. The latter ends the *voting period*, setting the *inVotingPeriod* variable to false. Then it clears proposals array in line 116 and resets votes mapping in lines 117-119.

```
98
99      /*
100         This method starts the voting period
101         and reset canVote array.
102     */
103     function _startVotingPeriod() internal{
104         inVotingPeriod = true;
105         for(uint256 i; i<voters.length;i++){
106             canVote[voters[i]] = true;
107         }
108     }
109
110     /*
111         This method ends the voting period and reset
112         proposals array and votes array.
113     */
114     function _endVotingPeriod() internal{
115         inVotingPeriod = false;
116         delete(proposals);
117         for(uint8 i; i<10;i++){
118             votes[i] = 0;
119         }
120     }
121
122 }
```

Issue description The `ErroneoudAccounting` contract has an *Erroneous Accounting Business Implementation* issue caused by lines 94-97 of the `endVoteAndRewards()` method:

```
94          treasury -= proposalReward;
95
96          _endVotingPeriod();
97      }
```

When the owner calls it, rewards are distributed among the winner *proposal* and *voters*. However, `treasury` is decreased only by `proposalRewards`. This operation doesn't take into account that also `voterReward` is removed from the contract, i.e., the amount is transferred to voter accounts.

So, after `endVoteAndRewards()` execution, the value inside `treasury` variable is different from the contract's balance, and the further voting rounds will compute rewards on a wrong `treasury` value.

1.8 L3 - Missing Business Logic Checks

Business Logic	**L3**	**Missing Business Logic Checks**
References	CWE	670 - Always-Incorrect Control Flow Implementation
	SWC-ID	110 - Assert Violation
	Demystifying	SE.2 - Unexpected environment or contract conditions

Vulnerability description This vulnerability could be hard to detect because it strongly depends on the business model and implementation. It is similar to *Erroneous accounting business implementation* (see Section 1.7), but instead of having a wrong model's formula implementation, there is a lack of checks, i.e., missing of `require()/assert()` statements that lead contract business logic to move away from the intended one. To explain this issue better, let's proceed with an example written by us:

```
// SPDX-License-Identifier: MIT
pragma solidity ^0.8.0;

contract Missing{

    uint256 public balance;

    struct positionStruct {
        uint256 amount;
        uint256 lastWithdrawOrClaimedRewards;
    }

    // An address is associated with a positionStruct. This
        is a "position"
    mapping(address => positionStruct) public position;

    /*  stake function can be used to stake funds to this
        contract.
    */
    function stake() public payable {
        if(position[msg.sender].amount == 0) position[msg.
            sender].lastWithdrawOrClaimedRewards = block.
            timestamp;
        position[msg.sender].amount += msg.value;
        balance += msg.value;
    }

    /*  withdraw function can be used to withdraw a funds
       amount from this contract.
        @parameter amount: the amount a user would to
           withdraw
    */
    function withdraw(uint256 amount) public{
        require(position[msg.sender].amount >= amount);
        position[msg.sender].amount -= amount;
        balance -= amount;
        payable(msg.sender).transfer(amount);
    }

    /*
        This function permits the staker to claim rewards.
           Staker will obtain 1 wei for every minute that
           has passed since the last withdrawal or
        reward claim
    */
    function claimRewards() public returns(uint256){
        require(position[msg.sender].
           lastWithdrawOrClaimedRewards + 7 days < block.
           timestamp, "It is too early to claim reward");
        uint256 reward = getClaimableRewards();
        position[msg.sender].lastWithdrawOrClaimedRewards =
           block.timestamp;
        if(reward > balance) reward = balance;
        balance -= reward;
        payable(msg.sender).transfer(reward);
    }

    function getClaimableRewards() public view returns(
       uint256){
        require(position[msg.sender].amount > 0, "Your
           address has not stake position");
        return 100 * (block.timestamp - position[msg.sender].
           lastWithdrawOrClaimedRewards)/ (60);
    }
}
```

The `Missing` contract allows staking of funds. A user can send ETHs to it using `stake()` function, which is `payable`, and can withdraw his/her funds using `withdraw(uint256 amount)`. Furthermore, a user can earn ETHs for the time that he/she staked funds, according to the formula

$$reward = 100 \cdot T$$

where T is the period since the last time the user withdrew or claimed the reward, expressed in minutes. If `reward` is higher than `Missing` balance, the user will obtain all funds of the contract, emptying it. The problem is that the user will not obtain the rewards that he/she deserves. `Missing` contract will give him/her all funds and will restart the user's `lastWithdrawOrClaimedRewards`. So the user will lose the remaining rewards.

This contract should have an additional check inside `claimRewards()` function, that makes transaction reverts if `Missing` contract does not have enough funds to reward the user. An example of a new `claimRewards()` function:

```
/*
    This function permits the staker to claim rewards. Staker
         will obtain 1 wei for every minute that has passed
        since the last withdrawal or
    reward claim
*/
function claimRewards(bool force) public returns(uint256){
    require(position[msg.sender].lastWithdrawOrClaimedRewards
         + 7 days < block.timestamp, "It is too early to
        claim reward");
    uint256 reward = getClaimableRewards();
    position[msg.sender].lastWithdrawOrClaimedRewards = block
        .timestamp;
    require(balance >= reward || force, "There are not enough
         funds. Set force parameter to claim reward anyway");
    balance -= reward;
    payable(msg.sender).transfer(reward);
}
```

In this way, it will be a user's decision if he/she wants or not to lose rewards.

1.9 L4 - Wrong Implementation

Business Logic	**L4**	**Wrong Implementation**
References	CWE	684 -Incorrect Provision of Specified Functionality
	SWC-ID	-
	Demystifying	-

Vulnerability description This issue describes when a smart contract implementation differs from the business model, e.g., when a function doesn't return the right value according to the business model, or when the project uses its standard or a community standard wrongly. Sometimes, the business logic model establishes standardization, and developers have to implement code that complies with standard definitions. If they don't, the future version of DApp, extension, and external contracts that rely on non-standard ones will have problems. This issue is quite generic and groups all vulnerabilities due to an incorrect implementation of the business model or the DApp internal standardization.

1.10 S1 - Reentrancy

Solidity	**S1**	**Reentrancy**
References	CWE	841 -Improper Enforcement of Behavioral Workflow
	SWC-ID	107 - Reentrancy
	Demystifying	L1 - Reentrancy

Vulnerability description *Reentrancy* is a well-known problem [9]. It is considered so serious that OpenZeppelin has deployed an apposite util module, called ReentrancyGuard. There are several guides and papers about this issue. According to [10], we can describe *Reentrancy* as a synchronization problem. We describe this issue using two contracts, `Victim` and `Attacker`:

```
Contract Victim{
    // This variable holds the balance of each address
    mapping (address => uint) public balanceOf;

    // An address can deposit eth to this contract, using
        this function
    // Victim's variable balanceOf is increased with the sent
         amount.
    function deposit() public payable {
        balanceOf[msg.sender] += msg.value;
    }

    // An address (msg.sender) can withdraw its eth using
        this function.
    // Function check if that address has enough deposited
        eth,
    // then perform a generic external call and
    // finally decrease msg.sender's balance by the amount
        withdrawn.
    function withdraw(uint amount) public {
        require(balanceOf[msg.sender] >= amount);
        msg.sender.call{value: amount}("");
        balanceOf[msg.sender] -= amount;
    }
}
```

```
Contract Attacker{
    // Assume that the attacker has an initial balance of 1
        wei.

    address victim_address = "victim_address"

    function attack() public{
        Victim(victim_address).deposit{value: address(this).
            balance}();
        Victim(victim_address).withdraw(1);
    }

    receive() external payable {
        if (Victim(victim_address).balance >= 1 wei) {
            Victim(victim_address).withdraw(1 wei);
        }
    }
}
```

We are using pseudocode, in part, but let's describe the attack vector. When the attacker calls `attack()` function of `Attacker` contract, he/she deposits the `Attacker balance` (that we assume has the initial value of 1 wei) and then calls the `withdraw` function of the `Victim` contract, passing 1 wei as `amount` parameter. `Victim.withdraw()` checks that the attacker has enough balance and then perform `msg.sender.callvalue: amount("");` This is an external call, that triggers the `Attacker.receive()` function. The attacker has maliciously written the `Attacker.receive()` function to call again `Victim.withdraw()`. The attacker will be able to perform many and many successful calls to `Victim.withdraw()` before the value of `balanceOf[msg.sender]` decrease.

Figure 1 visually describes the attack vector, where the Victim contract is called `Bank`.

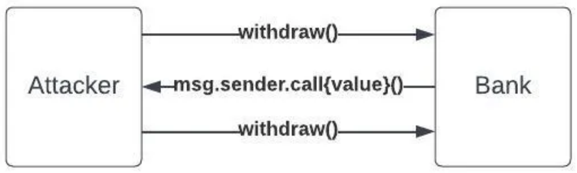

Fig. 1: Type of Cryptoasset, illustration from the article "The Ultimate Guide To Reentrancy" on Medium.com [10]

1.11 S2 - Precision Loss

Solidity	**S2**	**Precision Loss**
References	CWE	682 - Incorrect Calculation
	SWC-ID	-
	Demystifying	L2 - Rounding issues or precision loss

Vulnerability description EVM doesn't support floating-point arithmetic. Furthermore, compilers of Solidity version 0.4.24 and above neither support floating-point nor fixed-point data types. According to [11], it was a design decision rooted in determinism and predictability. In many other programming languages, many common issues about float numbers are well-known and documented[6].

Because usually contracts are used to perform complicated math operations, rounding errors and precision losses can happen. When somebody exploits them, he/she might be able to do much damage.

Let's go deeper to understand how this issue works:

```solidity
// SPDX-License-Identifier: MIT
pragma solidity 0.8.21;

contract precision_loss_example{
    uint256 public userReward;

    function example_1() public {
        uint256 userBalance = 150;
        uint256 totalLiquidity = 1000;
        uint256 poolReward = 100;

        userReward = (userBalance * poolReward) /
            totalLiquidity; // Results in 15, not 15.75
    }

    function example_2() public {
        uint256 userBalance = 150;
        uint256 totalLiquidity = 1000;
        uint256 poolReward = 100;

        userReward = (userBalance / totalLiquidity) *
            poolReward; // Results in 0, not 15.75
    }
}
```

[6] https://cwe.mitre.org/data/definitions/1077.html

In `example_1`, a value of 0.75 is completely cut off. Let's think if this means losing 0.75 ETH: it would be a huge loss. To avoid this, developers use less significant digits of integer numbers to describe the decimal part. For example, USDC stablecoin, defines 6 decimal digits. This means that a balance of 10^6 inside USDC ERC20 smart contracts represents the possession of 1 USDC. In `example_2`, the division is made before multiplication. Solidity computes it in this way:

$$\frac{userBalance}{totalLiquidity} = \frac{150}{1000} = round(0.15) = 0$$

In this case, the loss could be more impacting. There are several techniques to avoid this issue. First of all, usually, it is better to perform multiplication operations before division ones. Even if we think that mathematically certain formulas are the same, in Solidity they could not work. To avoid this issue, beyond scale members of a formula by a factor of 10^x, like in USDC, developers can use several methods. They can track fractions and accumulate them when they reach a whole unit, use some fixed-point libraries, or implement assertive checks after calculations.

1.12 S3 - Integer Overflow and Underflow

Solidity	**S3**	**Integer Overflow and Underflow**
References	CWE	682 - Incorrect Calculation
	SWC-ID	101 - Integer Overflow and Underflow
	Demystifying	L7 - Integer Overflow and Underflow

Vulnerability description Integer Overflow and Underflow are common problems in programming languages. These issues happen when an arithmetic operation on a variable moves its value out from the expected range. For example, in Solidity, the type `uint` has range $[0 : 2^{256} - 1]$. So, if we try to decrease by 1 a `uint` variable with value 0, that variable underflows, and its value will be $2^{256} - 1$

In an EVM environment, where numbers can represent money, this issue might cause critical loss. Thanks to this vulnerability, an attacker could bypass contract checks and perform forbidden operations. To avoid this issue, OpenZeppelin in 2018 developed the SafeMath library. If a contract implements SafeMath, any overflow/underflow will cause a revert of execution. From version 0.8.0 of the Solidity compiler, SafeMath is implemented by default. Of course, these additional checks cost gas. For this reason, it was introduced a new solidity construct: `unchecked`. Only an arithmetic operation inside `unchecked` can overflow/underflow without causing the transaction to revert. Let's see an example[7] of this issue, using an old version of the solidity compiler:

```
pragma solidity ^0.4.21;

contract TokenSaleChallenge {
    mapping(address => uint256) public balanceOf;
    uint256 constant PRICE_PER_TOKEN = 1 ether;

    function TokenSaleChallenge(address _player) public
        payable {
        require(msg.value == 1 ether);
    }

    function isComplete() public view returns (bool) {
        return address(this).balance < 1 ether;
    }

    function buy(uint256 numTokens) public payable {
        require(msg.value == numTokens * PRICE_PER_TOKEN);

```

[7] https://capturetheether.com/challenges/math/token-sale/

```
        balanceOf[msg.sender] += numTokens;
    }

    function sell(uint256 numTokens) public {
        require(balanceOf[msg.sender] >= numTokens);

        balanceOf[msg.sender] -= numTokens;
        msg.sender.transfer(numTokens * PRICE_PER_TOKEN);
    }
}
```

There are two main functions: `buy(uint256 numTokens)` and `sell(uint256 numTokens)`. The `balanceOf` variable maps users' addresses to `uint256`, which represents their balances.

Let's try to overflow `balanceOf[msg.sender]` using `buy(uint256 numTokens)`. Take a look at this `require`:

```
require(msg.value == numTokens * PRICE_PER_TOKEN);
```

It checks the value sent by the caller. Analyzing this line, we obtain:

$$msg.value = numTokens \cdot PRICE\_PER\_TOKEN \tag{1}$$
$$msg.value = numTokens \cdot 1ether \tag{2}$$
$$msg.value = numTokens \cdot 10^{18} > numTokens \cdot 2^{59} \tag{3}$$

If $numTokens \cdot 2^{59}$ overflows, also $numTokens \cdot PRICE\_PER\_TOKEN$ will overflow. $numTokens \cdot 2^{59}$ overflows when:

$$numTokens \cdot 2^{59} > 2^{256} - 1 \tag{4}$$

So `numTokes` should be equal to or greater than α, where α is:

$$\alpha = \left\lceil \frac{2^{256} - 1}{2^{59}} \right\rceil \cong 2.00867255532 \cdot 10^{59} \tag{5}$$

When $numTokens \geq \alpha$, an overflow will occur. Using $numTokens = \alpha$:

$$numTokens \cdot PRICE\_PER\_TOKEN = \alpha \cdot 10^{18} \mod 2^{256} \tag{6}$$
$$numTokens \cdot PRICE\_PER\_TOKEN = 415992086870360064 \tag{7}$$

If we call `buy(uint256 numTokens)` using α as `numTokens` and $msg.value = 415992086870360064 = 0.415992086870360064\ ether$, we will respect require check, and our balance will increase by an α amount, even if we didn't own that amount before. In general, we can successfully call `sell(uint256 numTokens)` using any `numTokens` value between $[0 : \alpha]$ i.e $[0 : 2 \cdot 10^{59}\ ether]$. We underline that using the same attack vector, with the same contract but with a compiler version greater than 0.8.0, the transaction reverts.

After SafeMath introduction, another problem has come up: *Integer Overflow and Underflow DoS*. In fact, in some situations, a contract could have a math operation that always reverts and prevents any corrective action. An attacker could be able to exploit this and block the contract's functionalities.

1.13 S4 - Using `tx.origin`

Solidity	**S4**	**Using `tx.origin`**
References	CWE	477 - Use of Obsolete Function
	SWC-ID	115 - Authorization through tx.origin
	Demystifying	L11 - Using tx.origin

Vulnerability description `tx.origin` and `msg.sender` are two special variables. `tx.origin` is a special variable in solidity that returns the address of the initial caller, i.e. the EOA address that originated the transaction. `msg.sender` is another special variable that returns the address of whatever triggered directly current execution. Using `tx.origin` to check the caller address (e.g. to verify that an address is the owner of the contract) can be exploited by an attacker. Usually, the exploit is made through phishing, convincing the victim EOA to execute a transaction to the attacker contract. Let's see an example[8]:

```
pragma solidity ^0.8.0;

contract Wallet {
    address public owner;

    constructor() payable {
        owner = msg.sender;
    }

    function transfer(address payable _to, uint _amount)
        public {
        require(tx.origin == owner);

        (bool sent, ) = _to.call{value: _amount}("");
        require(sent, "Failed to send Ether");
    }
}
```

This contract use `transfer(address payable _to, uint _amount)` to send `_amount` to `_to` address. Before transferring the amount, this function checks that `tx.origin == owner`. In this example, we'll ignore gas fees. In Table 2 we report the naming conventions we used in this issue description.

In Table 3 we reported the conventions in name balances and their values over time. *T=0* is the initial situation. *T=1* is the situation just after Bob deploys `Wallet` smart contract. *T=2* is the final situation, after the exploit. After Bob have deployed the `Wallet` smart contract on blockchain (T=1), Eve deploys the following smart contract:

[8] `https://solidity-by-example.org/hacks/phishing-with-tx-origin/`

Roles and Contracts	Addresses
B: Bob, the victim	$B_{address}$: Bob EOA
E: Eve, the attacker	$E_{address}$: Eve EOA
W: Wallet, the exploitable contract	$W_{address}$: address of deployed wallet contract
A: Attacker, the attacker contract	$A_{address}$: address of deployed attacker contract

Table 2: Name conventions used in this section.

		Values of balances over time		
Name	Description	T=0	T=1	T=2
$B_{balance}$	balance of Bob EOA	1 Eth	0	0
$E_{balance}$	balance of Eve EOA	0	0	1 Eth
$W_{balance}$	balance of deployed wallet contract	0	1 Eth	0
$A_{balance}$	balance of deployed attacker contract	0	0	0

Table 3: Balances name conventions and values over time.

```
pragma solidity ^0.8.0;

interface Wallet{
    function transfer(address payable _to, uint _amount)
        external;
}

contract Attacker {
    address payable public owner;
    Wallet wallet;

    constructor(address wallet_address) payable {
        owner = payable(msg.sender);
        wallet = Wallet(wallet_address);
    }

    receive() external payable {
        wallet.transfer(owner, address(wallet).balance);
    }
}
```

Eve uses $W_{address}$ as input of the `constructor`. So, after A deployment, the `onwer` of A is Eve and $A_{balance}$ is 0. Then, Eve convinced Bob to send 1 Wei to Eve's contract A. When Bob calls this transaction, he triggers `receive()` function of A.

`Attacker.receive()` calls `Wallet.transfer()` using $E_{address}$ and $W_{balance}$ as input parameters: in this call, $tx.origin = B_{address}$. This call doesn't revert and W's *transfer()* successfully transfer the whole $W_{balance}$ to $E_{address}$. In Table 3 there are values at *T=2* after the vulnerability is exploited: Eve stole 1 Eth from the $W_{balance}$.

1.14 S5 - Arbitrarily Code Injection

Solidity	**S5**	**Arbitrarily Code Injection**
References	CWE	829 - Inclusion of Functionality from Untrusted Control Sphere
	SWC-ID	-
	Demystifying	L6 - Arbitrary external function call

Vulnerability description In smart contracts, developers usually import functionalities from other contracts. A common way to do so is using interfaces. A developer can *wrap* an address with an interface if such address implements at least the functionalities defined in that interface.

Developers might allow wrapping an interface around an address supplied by the user. An attacker could create a contract that implements interface methods but with malicious code inside, and then pass the address of the malicious contract to the victim contract. The same could happen using `call` on the external address that can be manipulated by an attacker. A common way to avoid this is using an *allow-list*, where developers can insert only addresses they trust. Let's see an example of the vulnerable contract written by us:

```solidity
// SPDX-License-Identifier: MIT
pragma solidity ^0.8.0;

interface DepositTracker{
    function addAmount(uint256 amount) external;

    function getAmount() external view returns(uint256);
}

contract Victim {
    constructor(address depositTrackerAddress) payable{
        DepositTracker depositTracker = DepositTracker(depositTrackerAddress);
        depositTracker.addAmount(msg.value);
    }

    function withdrawAll(address depositTrackerAddress) public{
        DepositTracker depositTracker = DepositTracker(depositTrackerAddress);
        uint256 amount = depositTracker.getAmount();
        payable(msg.sender).transfer(amount);
    }
}
```

Bob is a Solidity developer. He has created the `Victim` contract. He wants this contract to keep his money in its balance, tracking the balance amount in another contract that he has already deployed, `depositTracker`:

```
// SPDX-License-Identifier: MIT
pragma solidity ^0.8.0;

contract DepositTracker{
    mapping(address => uint256) public track;

    function addAmount(uint256 amount) public {
        track[msg.sender] += amount;
    }

    function getAmount() public view returns(uint256){
        return track[msg.sender];
    }
}
```

So, when Bob deployed the `Victim` contract, he passed the `depositTracker` address and 1 gwei as `msg.value`. He can withdraw all his funds by calling the `Victim.withdrawAll()` function and passing the `depositTracker` address as a parameter. Let's analyze the `Victim.withdrawAll()` function:

```
function withdrawAll(address depositTrackerAddress) public{
    DepositTracker depositTracker = DepositTracker(
        depositTrackerAddress);
    uint256 amount = depositTracker.getAmount();
    payable(msg.sender).transfer(amount);
}
```

This function wraps around `depositTrackerAddress` address the `DepositTracker` interface, and then call `depositTracker.getAmount()`. Eve saw the issue, and deployed a new contract:

```
// SPDX-License-Identifier: MIT
pragma solidity ^0.8.0;

contract fakeDepositTracker{

    address victim;

    constructor(address _victim){
        victim = _victim;
    }

    function addAmount(uint256 amount) public {}

    function getAmount() public view returns(uint256){
        return victim.balance;
    }
}
```

This contract maliciously implements its version of `getAmount()`, which doesn't return Eve's tracked balance, but the Victim's balance. Then, Eve deploys this contract and calls `Victim.withdrawAll()`, passing `fakeDepositTracker` address as parameter. Thanks to this, Eve obtained all of Bob's funds.

1.15 S6 - Outdated Compiler Bug

Solidity	**S6**	**Outdated Compiler Bug**
References	CWE	937 - Using Components with Known Vulnerabilities
	SWC-ID	102 - Outdated Compiler Version
	Demystifying	-

Vulnerability description We define this vulnerability to group all issues that are the consequence of an outdated compiler/optimizer version. We underline that assembly code inside contracts could be particularly vulnerable. We don't go deeper into this kind of vulnerability. All vulnerabilities that are well-known for a specific solidity version range, should be considered in this category.

1.16 S7 - Unsafe Casting

Solidity	**S7**	**Unsafe Casting**
References	CWE	704 - Incorrect Type Conversion or Cast
	SWC-ID	-
	Demystifying	-

Vulnerability description This vulnerability appears in many programming languages. Solidity allows *casting* variables, i.e., converting one type to another, both explicitly and implicitly. For example, an `uint64` can be converted implicitly in an `uint256`. However, inverse conversion needs to be done explicitly:

```
// implicit casting
uint64 a = 100;
uint256 b = a;

// explicit casting
uint256 c = 100;
uint64 d = uint64(c);
```

When an explicit casting is applied, some more significant bits could be deleted. Let's do an example with `uint8` and `uint16`. We remind that $max(uint8) = 2^8 - 1 = 255$. The following are examples of `uint8` casting:

```
uint16 a = 255;
uint8 b = uint8(a); // b is equal to 255

uint16 a = 256;
uint8 b = uint8(a); // b is be equal to 0

uint16 a = 257;
uint8 b = uint8(a); // b is be equal to 1
```

Let's see a simple example that explains how this could be exploited by an attacker. Bob wrote a contract. He thinks that only those who will guess `key` value will obtain the reward. However, everybody can read contracts on blockchain, and then infer `key` right value.

```
// SPDX-License-Identifier: MIT
pragma solidity ^0.8.0;

contract VeryStrongLock{

    constructor() payable{}

    function lock(uint256 key) public {
        if(uint128(key) == 23){
            payable(msg.sender).transfer(1 gwei);
        }
    }
}
```

Bob understands the problem before deploying the contract on the blockchain, and decides to modify it with the following instruction:

```
// SPDX-License-Identifier: MIT
pragma solidity ^0.8.0;

contract VeryStrongLock{

    constructor() payable{}

    function lock(uint256 key) public {
->      require(key > 2<<127, "key is too small"); // key
    have to be greater than 2^128

        if(uint128(key) == 23){
            payable(msg.sender).transfer(1 gwei);
        }
    }
}
```

Now, he thinks that this will be a secure lock, because $23 << 2^{128}$. Unfortunately, Bob forgot `uint128` explicit casting. When Eve sees the contract, she understands immediately the problem. To obtain payment, she has to use `key` such that:

$$key \mapsto key - 2^{128} = 23$$

So, she calls `lock` function passing $2^{128} + 23$ and obtains Bob's funds.

1.17 S8 - Using `transferFrom()` instead of `safeTransferFrom()` in ERC721 implementation

Solidity	**S8**	**Using `transferFrom()` instead of `safeTransferFrom()` in ERC721 implementation**
References	CWE	684 - Incorrect Provision of Specified Functionality
	SWC-ID	-
	Demystifying	-

Vulnerability description This issue is related to the *EIP721* NFT standard. `transferFrom()` and `safeTransferFrom()` both transfer ownership of NFT with `_tokenId` from address `_from` to address `_to`:

```
transferFrom(address _from, address _to, uint256 _tokenId)
safeTransferFrom(address _from, address _to, uint256 _tokenId
    )
```

While `transferFrom()` executes just the transferring of the NFT ownership, `safeTransferFrom()` makes additional checks: it verifies that `_to` is a smart contract and then uses `onERC721Received` function on `_to` to be sure that `_to` address is the new `_tokenId` owner. Despite `safeTransferFrom()` being more gas-expensive thanks to these further checks, it is almost always safer to use it than `transferFrom()`. However, developers sometimes prefer to self-build the NFT transferring logic. In these cases, `transferFrom()` can be used, but developers should implement security checks anyway.

1.18 S9 - Using `_mint()` instead of `_safeMint()` in ERC721 implementation

Solidity	**S9**	**Using `_mint()` instead of `_safeMint()` in ERC721 implementation**
References	CWE	684 - Incorrect Provision of Specified Functionality
	SWC-ID	-
	Demystifying	-

Vulnerability description This issue is strongly related to the EIP721 NFT standard. According to EIP721 standard and OpenZeppelin ERC721, `_mint()` and `_safeMint()` both create a new NFT with unique `tokenId` and transfer ownership to address `to`:

```
_mint(address to, uint256 tokenId)}
_safeMint(address to, uint256 tokenId)
```

While `_mint()` function just creates the new NFT transferring it to `to` address, `_safeMint` make additional checks: it verifies that at address `_to` there exists a valid smart contract and then uses `onERC721Received` function on `_to` to be sure that `_to` address is the new `_tokenId` owner, i.e., the ownership of the minted NFT has been assigned to the right address. So, it is almost always safer to use `_safeMint` than `_mint`, even if the first is more gas expensive. However, sometimes developers prefer to self-build the NFT *mint* logic because they want to implement behaviors not supplied by the OpenZeppelin (or Solmate) framework. In these cases, they should also implement security checks on minting actions.

1.19 S10 - Dangerous `DelegateCall`

Solidity	**S10**	**Dangerous `DelegateCall`**
References	CWE	829 - Inclusion of Functionality from Untrusted Control Sphere
	SWC-ID	112 - Delegatecall to Untrusted Callee
	Demystifying	L6 - Arbitrary external function call
		L8 - Revert issues caused by low-level calls or external libraries

Vulnerability description This is a very well-known vulnerability [12][13]. It involves the `DELEGATECALL` opcode. This opcode can be called using `delegatecall()` function defined in Solidity: this permits to call an external function using the caller context. So, it has to be used carefully. When an attacker can use arbitrarily `delegatecall` using an external malicious contract, he/she can manipulate the contract's state. This is an example written by us:

```
// SPDX-License-Identifier: MIT
pragma solidity ^0.8.0;

contract DangerousDelegateCall{

    // first 18 digits of pi
    uint64 very_precise_pi = 3141592653589793238;

    // initial settings
    uint256 public min_diff = 2<<255 - 1;
    address public current_winner;

    /*
        This is the core function. This function calls _address.compute_pi() using delegatecall.
        If this call is successful, take the return value and compute the difference between it and the very_precise_pi value and if it is less than the leader one, current_winner is updated with
        msg.sender address.
    */
    function pi_calcolator(address _contract) public {
        (bool success, bytes memory data) = _contract.delegatecall(
            abi.encodeWithSignature("compute_pi()")
        );
        require(success, "Error in delegatecall");
23
24         if(data.length > 0){
25             uint256 computed_pi = abi.decode(data, (uint256));
26             uint256 diff;
27             if (very_precise_pi >= computed_pi){
28                 diff = very_precise_pi - computed_pi;
29             }
30             else{
31                 diff = computed_pi - very_precise_pi;
32             }
33             if(diff < min_diff){
34                 min_diff = diff;
35                 current_winner = msg.sender;
36             }
37         }
38     }
39 }
```

Bob has invented a game, and he writes `DangerousDelegateCall` contract to implement it. A participant has to write his/her contract with `compute_pi()` function, then ask through `pi_calcolator(address _contract)` function, that uses `delegateCall`, to execute it. If the participant has implemented a good function to compute the pi value, he/she becomes the new leader (i.e., the address saved into `current_winner` variable). `DangerousDelegateCall` has three state variables. In `very_precise_pi`, it is stored the first 19 digits of pi. In `min_diff` is stored the difference between the `vary_precise_pi` and the value computed by the current winner participant. The initial value is $2^{255} - 1$, i.e., the max value that can be stored in a `uint256` variable. In the `current_winner` variable, `DangerousDelegateCall` saves the address of the current winner. The initial value is `address(0)` (the default one).

Let's analyze the `pi_calcolator()` function. It receives an address as a parameter, then it calls the `compute_pi()` function deployed at that address using `delegatecall()` (lines 19-21). If `delegatecall()` succeeds, i.e., there exists a smart contract at address `_contract` that implements a `compute_pi()` function with correct signature, `compute_pi()` compute the difference between the value returned from `delegatecall()` and `very_precise_pi` (lines 25-32). If this difference is less than the value saved in `min_diff` state variable, the caller, i.e., the `msg.sender` address, becomes the new winner address (lines 33-36).

Alice is an honest player and wants to participate in Bob's game. She writes her contract with `compute_pi()` function:

```
// SPDX-License-Identifier: MIT
pragma solidity ^0.8.0;

contract GoodPiCalcolator{

    function compute_pi() public returns(uint256 pi){
        uint256 decimals = 18;
        uint256 iterations = 4000;
        uint256 base_value = 4*(10**decimals);
        pi = base_value;
        uint256 bottom = 3;
        for(uint256 i=0; i< iterations; ++i){
            if(i%2 == 0){
                pi = pi - (base_value/bottom);
            }
            else{
                pi = pi + (base_value/bottom);
            }

            bottom = bottom + 2;
        }
    }
}
```

She is good in maths and decides to create a function that implements the Leibniz Formula[9] with 4000 iterations. She deploys `GoodPiCalcolator` and then passes its address to`DangerousDelegateCall.pi_calcolator()`. It's a very good result:

```
very_precise_pi = 3141592653589793238
pi approximation computed by Alice = 3141842591101510998
difference = 249937511717760 (0,0079259% of very_precise_pi)
```

Eve wants to participate in Bob's game and ruin it. She understands that she can exploit `delegatecall`. So, she writes her contract with `compute_pi()` function:

```
// SPDX-License-Identifier: MIT
pragma solidity ^0.8.0;

contract MaliciousPiCalcolator{

    function compute_pi() public{
        selfdestruct(payable(address(this)));
    }
}
```

She knows that when `DangerousDelegateCall` uses `delegatecall` to execute the `compute_pi()` inside `MaliciousPiCalcolator` contract, it runs that function using the context of the `DangerousDelegateCall` contract. So `selfdestruct()` function destroys `DangerousDelegateCall`.

[9] https://en.wikipedia.org/wiki/Leibniz_formula_for_%CF%80

1.20 S11 - Relying on External Source

Solidity	**S11**	**Relying on External Source**
References	CWE	829 - Inclusion of Functionality from Untrusted Control Sphere
	SWC-ID	113 - DoS with Failed Call
		104 - Unchecked Call Return Value
	Demystifying	L8 - Revert issues caused by low-level calls or external libraries
		S1 - Price oracle manipulation

Vulnerability description This issue occurs when there is a strong bonding of the vulnerable contract with an external source. Usually, developers write code that interacts with external sources to obtain services or information. For example, a contract could interact with external oracles to obtain some token price. Even if developers trust external sources, they should interact with them carefully.

We want to remind the reader that once a contract is deployed on the blockchain, it could be hard to modify it. Developers should always write code thinking that the external source could be hacked and work abnormally or doesn't work at all. So, they should implement integrity checks of external source data. In the worst case, a contract should be able to remove the compromised data source, even using some governance mechanisms.

We describe the vulnerability using an example written by us:

```solidity
// SPDX-License-Identifier: MIT
pragma solidity ^0.8.0;

interface Oracle {
    function getPrice(string memory token1, string memory
        token2) external returns(uint256);
}

contract RelyingOnExternalSource{
    address[] oracles = [// list of 4 addresses];

    // weights to compute the weighted arithmetic mean
    mapping(address => uint256) weight;

    // We assign weights inside the constructor function
    constructor() {
        weight[oracles[0]] = 1;
        weight[oracles[1]] = 5;
        weight[oracles[2]] = 8;
        weight[oracles[3]] = 10;
    }

    // This function changes weight. In case of a compromised oracle, it will
    // possible to change its weight to 0
    function changeWeight(address _address, uint256 _newValue) public{
        weight[_address] = _newValue;
    }

    /*
        Oracle_1.price = 2038   Oracle_1.weight = 1
        Oracle_2.price = 2000   Oracle_1.weight = 5
        Oracle_3.price = 2080   Oracle_1.weight = 8
        Oracle_4.price = 2020   Oracle_1.weight = 10

        computePrice() = (2038*1 + 2000*5 + 2080*8 + 2020*10)/24 = 2036,58333
    */
    function computePrice() public returns(uint256 price){
        uint256 total_weight = 0;
        for(uint256 i=0; i<oracles.length; i++){
            total_weight += weight[oracles[i]];
        }

        uint256 total_price = 0;
        uint256 oracle_price = 0;
        for(uint256 i=0; i<oracles.length; i++){
            oracle_price = Oracle(oracles[i]).getPrice("ETH", "USDC");
            total_price += oracle_price * weight[oracles[i]];
        }

        price = total_price / total_weight;
    }
}
```

Then, we deploy 4 oracles. The first three using this code:

```
contract Oracle_1{
    function getPrice(string memory token1, string memory token2) public returns(uint256){
        return 2038;
    }
}

contract Oracle_2{
    function getPrice(string memory token1, string memory token2) public returns(uint256){
        return 2000;
    }
}

contract Oracle_3{
    function getPrice(string memory token1, string memory
        token2) public returns(uint256){
        return 2080;
    }
}
```

The last one represents the breakable external source. It returns the price only when it is not broken. Otherwise, it reverts:

```
contract BreakableOracle{

    bool broken = false;

    function breakOracle() public{
        broken = true;
    }
    function unbreakOracle() public{
        broken = false;
    }

    function getPrice(string memory token1, string memory
        token2) public returns(uint256){
        require(!broken);
        return 2020;
    }
}
```

When all oracles work correctly, `RelyingOnExternalSource.computePrice()` works fine and returns `2036`, according to *weighted arithmetic mean*[10].

When `BreakableOracle` goes down, developers try to set its weight to 0 using `RelyingOnExternalSource.changeWeight()`. However, it will not work. Even if `BreakableOracle` doesn't contribute to the mean, its `getPrice()` function is still called, reverting. So, without a good corrective mechanism, a contract that relies on an external source can stop working when that source goes down.

[10] `https://en.wikipedia.org/wiki/Weighted_arithmetic_mean`

1.21 S12 - Division by Zero DoS

Solidity	**S12**	**Division by Zero DoS**
References	CWE	667 - Improper Locking
	SWC-ID	-
	Demystifying	L3 - Bugs that are caused by using uninitialized variables
		S3 - Erroneous state updated

Vulnerability description This issue is a kind of DoS. It is a consequence of an erroneous implementation of a formula or an unprotected modification of a variable that is used as a denominator in a division.

Like other programming languages, Solidity doesn't accept division by 0. However, it could happen thanks to an erroneous implementation of the code. When this issue occurs, the transaction reverts. There are two main root causes for this issue:

– denominator variable is uninitialized, i.e. no line of code sets its value, or there are some paths in execution where the variable is not set. In this case, an attacker could try to make these paths the only ones available, causing a DoS.
– denominator variable has been initialized, but some public functions are manipulable by an attacker, that can change the state of the contract to set the variable value to zero, and avoid changing it indefinitely

Until now, describing this issue, we didn't make a distinction between state and local variables. Anyway, for the first case, the attack could rely both on state and local variables. For the second, the attacker should be able to manipulate a state variable.

Let's see an example of the second case written by us:

```
// SPDX-License-Identifier: MIT
pragma solidity ^0.8.0;

contract DivisionByZero{
    uint256 public developerReward = 0;
    uint256 public percentageNumerator = 1;
    uint256 public percentageDenominator = 100;

    uint256 public lastAmount; // Amount sent by the last
        participant that called participate() successfully

    address public developerAddress;
    address public currentWinner;

    uint256 public endTimestamp; // Timestamp of the moment when the contest ends.

    constructor() payable{
        developerAddress = msg.sender;
        endTimestamp = block.timestamp + 1 days;
    }

    /*
    A user can participate in the contest using this function.
    @param newPercentageNumerator, newPercentageDenominator: numerator and denominator for new developerReward
        percentage, that will be applied to the next currentWinner.
    */
    function participate(uint256 newPercentageNumerator, uint256 newPercentageDenominator) public payable returns(uint256){
        uint256 toDeveloperReward = (msg.value * percentageNumerator)/percentageDenominator;
        require(toDeveloperReward < msg.value); // Check to avoid 100% percentage to developerReward
        uint256 toContestReward = msg.value - toDeveloperReward;

        require(msg.value - toDeveloperReward > lastAmount, "The new amount must be greater than last participant amount");

        developerReward += toDeveloperReward;
        currentWinner = msg.sender;
        percentageNumerator = newPercentageNumerator;
        percentageDenominator = newPercentageDenominator;
        lastAmount = msg.value;
    }

    /*
    This function can be called by everybody. However, it will be executed successfully only 24h to contest starts.
    */
    function closeContest() public{
        require(block.timestamp > endTimestamp);
        payable(developerAddress).transfer(developerReward);
        payable(developerAddress).transfer(address(this).balance);
    }
}
```

In `DivisionByZero` contract, Bob has implemented a simple gambling game in which everyone can participate. The game has the following rules:

– A user can participate successfully and become the current winner if he/she sends to `DivisionByZero` contract an amount of ETH larger than the last one. However, this amount is computed by subtracting a fee that will be sent to developers (Bob) when the contest finishes.
– The sum of all fees is saved in `developerReward` variable. When a user participates successfully in the game, a new fee is computed using the two values inside `percentageNumerator` and `percentageDenominator` variables that have been set by the last `currentWinner` and added to `developerReward` variable. The initial fee percentage is 1%;
– `developerReward` percentage can't be equal to 100%. Otherwise, a user could prevent other users from participating.
– Contest duration: 24 hours

After Bob has finished writing the contract, he deploys it on the blockchain, triggering the execution of the `constructor()` function, which is `payable`. So, Bob can deploy `DivisionByZero` contract sending 1 ETH as an initial reward.

Eve likes Bob's game and wants to participate. Since Bob launched the game, several players participated in sending their ETH, and now there are 15 ETH in the contract's balance, of which 5 ETH in `developerReward`. All previous participants decided to not change the `developerReward` percentage, which is still 1%: its value is computed using the formula `percentageNumerator/percentageDenominator` at line 27. The initial values of these two variables were set when the contract was deployed (lines 6-7).

The current winner sent 3 ETH, and this value has been saved into `lastAmount` variable. To become the new current winner, she sends 3.04 ETH, because

$$(3.04 \cdot 10^{18}) \cdot 99\% > 3 \cdot 10^{18} = lastAmount$$

Thanks to the fact she is sending enough ETH, she participates successfully and obtains the right to define the new values for `percentageNumerator` and `percentageDenominator` variables. She chooses the following values:

```
percentageNumerator = 1
percentageDenominator = 0
```

After that, Eve will be sure that nobody else can be `currentWinner` because every call of `participate()` function will revert thanks to the *division by zero* exception. This is because of this line:

```
30        uint256 toDeveloperReward = (msg.value *
              percentageNumerator)/percentageDenominator;
```

where the value of `percentageDenominator` variable will remain zero indefinitely.

1.22 S13 - Temporal DoS

Solidity	**S13**	**Temporal DoS**
References	CWE	667 - Improper Locking
	SWC-ID	-
	Demystifying	S3 - Erroneous state updated

Vulnerability description This issue is another kind of DoS. In Solidity, there are mainly two special variables that are useful to manage temporal requirements inside the contract: `block.number` and `block.timestamp`. Developers can write code that depends on these special variables. However, they should use them carefully, because the values of these variables strongly depend on blockchains and they can be foreseen by an attacker, that could exploit this fact using, for example, a *Frontrunning* attack (see Section 1.1). This vulnerability relies on the fact that an attacker could arbitrarily modify a contract's state and force a contract to an execution path that always reverts for an indefinite period, causing the DoS of the contract.

We want to describe this issue with another example written by us. Bob wants to make a new game after Eve ruined the last ones. While the contest is open, everybody can participate by sending at least 1 gwei. The contest will close in 24 hours. The last user that will manage to participate will win a Bob's NFT that he has previously approved to `TemporalDoS` address. The contest finishes when anybody calls `closeContest` function after 24 hours from the deploying of `TemporalDos` contracts. When `closeContest` is successfully called, it will trigger NFT transferring to `CurrentWinner` address.

```
// SPDX-License-Identifier: MIT
pragma solidity ^0.8.0;

import "https://github.com/OpenZeppelin/openzeppelin-
    contracts/blob/master/contracts/token/ERC721/ERC721.sol";


contract TemporalDoS{

    address owner;
    uint256 NFT_id;
    address NFT_pool;
    uint256 min_deposit = 1 gwei;
    uint256 timer;
    address public currentWinner;
    uint256 endTimestamp;

    constructor(uint256 _NFT_id, address _NFT_pool) {
        owner = msg.sender;
        NFT_id = _NFT_id;
        timer = block.timestamp;
        currentWinner = msg.sender;
        endTimestamp = block.timestamp + 1 days;
        NFT_pool = _NFT_pool;
    }

    /*
    A user can participate in the contest using this function. To become the currentWinner,
    a participant has to send at least the min_deposit value.
    After a user participates successfully, the function resets the timer
    */
    function participate() public payable returns(uint256){
        require(msg.value >= min_deposit);
        require(block.timestamp > timer);
        timer = block.timestamp + 10 seconds;
        currentWinner = msg.sender;
    }

    /*
    This function can be called by everybody. However, it will be executed successfully only 24h to contest starts.
    When it will call successfully, it will transfer ownership of NTF_id to currentWinner
    */
    function closeContest() public{
        require(block.timestamp > endTimestamp);
        IERC721(NFT_pool).safeTransferFrom(owner, currentWinner, NFT_id);
    }
}
```

Eve likes Bob's games, and she wants to participate in the `TemporalDoS` contest. Other participants have already sent more than 1 ETH. She understands that she can avoid others from participating. She calls `participate()` function and becomes `CurrentWinner`. After that, she starts to call `participate()` function systematically every 10 seconds, thanks to a script she wrote. In this way, `timer` variable is always reset, and nobody else can participate.

Even if there is a small possibility that another user succeeds in participating once, Eve could increase gas fees and make this more unlikely. Of course, this attack is costly (gas fees and many deposit calls). However, Eve starts her attack only the last 10 minutes before the contest ends. In this way, she succeeds in winning Bob's NFT.

1.23 S14 - Erroneous ERC721 Implementation

Solidity	**S14**	**Erroneous ERC721 Implementation**
References	CWE	684 - Incorrect Provision of Specified Functionality
	SWC-ID	-
	Demystifying	S2 - ID-related violations

Vulnerability description This issue represents several issues that can happen during *ERC721* implementation. This standard is complex and could lead to many errors. We have already described NFT and *ERC721*. The main characteristic is that every NFT is unique, i.e. it has a unique ID inside the *ERC721* implementation. This is the big difference to fungible tokens, which are indistinguishable from each other. Another common error during *ERC721* implementations is written code that doesn't check if a certain NFT ID exists or not.

1.24 S15 - Erroneous `Pausable` implementation

Solidity	**S15**	**Erroneous `Pausable` implementation**
References	CWE	684 - Incorrect Provision of Specified Functionality
	SWC-ID	-
	Demystifying	-

Vulnerability description This issue is strictly correlated with OpenZeppelin Pausable implementation. A contract can be defined as *Pausable*, i.e., inherits state variables and methods from OpenZeppelin Pausable contract. When that contract is paused, all methods with `WhenNotPaused` modifier will revert if called. This mechanism can be useful to stop all contract functionalities when, for example, the contract is under attack, or developers notice that it has a vulnerability.

Usually, `pause()` and `unpause()` methods are public and callable only by special roles. However, some implementations of *Pausable* contracts (for example, security/Pausable.sol in the openzeppelin-contracts-upgradeable library, release v4.8) implement only internal methods, i.e., implement only `_pause()` and `_unpause()`. In these cases, once deployed contracts, there will be no way to call these methods. Developers have to remember to implement public methods that call the internal ones. Pausable implementation is a key feature of modern smart contract development because allows for blocking malicious behavior freezing contracts' methods.

2 Dataset overview

We mainly used contracts from the C4 contest to create our dataset. We can build a solid truth base in this way because every contract in these contests has been analyzed deeply and the warden's reports were judged by C4's experts.

DApp	Contest	Contracts	Abstract contracts	Libraries	Interfaces	SLOCs
Ajna	2023/05	3	3	1	4	1391
Asymmetry	2023/03	4	0	0	0	460
Caviar	2023/04	4	0	0	1	741
Eigenlayer	2023/04	7	2	3	12	1393
ENS	2023/04	9	0	9	0	2022
Frankencoin	2023/04	8	2	0	0	949
Juicebox	2023/05	1	0	0	0	160
Livepeer	2023/08	4	1	2	3	1605
Llama	2023/06	11	4	2	3	2047
Shell Protocol	2023/08	1	0	0	0	460
Stader Labs	2023/06	21	0	1	0	4334
	Overall	73	12	18	23	15562

Table 4: Dataset Overview.

In Table 4 we report details of chosen C4 contests. We analyzed 11 contests from March to August 2023. In detail, we run selected tools over 73 `contracts`, 126 objects, and 15.562 SLOCs. Analyzed contests cover a wide range of categories, from *Lending* to *Derivative* DApps, from *Stablecoin* to *Social* Protocols. In the remainder of this section, we provide more details about these contests. In particular, for each contest, there are 4 paragraphs:

– **DApp description**: in this section, we describe what DApp aims to do and the main DApp concepts;
– **C4 contest description**: in this section, we describe what contracts are in the scope of the C4 contest. Usually, sponsors ask to review only a part of the code: the part that they think is more critical or that has not been audited so far;
– **C4 report description**: in this section, there are 4 tables. First, we report an overview of wardens and bot races findings. The second and third tables detail the high and medium wardens' findings. Finally, the fourth table reports in detail the high and medium bot race winner findings. In this table,

we add a column labeled *valid*: due to the short period (24 hours) that the judge has to verify bot race findings, there may be several false positives. For each table, we report corresponding category IDs described in Section 1;

– **Analysis tools results**: in this section, we report findings of state-of-the-art automatic tools Mythril, Slither, and Smartcheck.

2.1 Ajna

Ajna				Lending DApp	
	Contracts	Abstract contracts	Libraries	Interfaces	Overall
#	3	3	1	4	11
SLOCs	604	540	38	209	1391
Pre-contest automatic tool analysis			Slither		

Table 5: Ajna's 2023-05 contest.

DApp description The Ajna protocol is described in [14]. In Table 5 there is an overview of the 2023-05 contest. Ajna is a lending and borrowing protocol.

Let's introduce some definitions from the protocol:

– **Pool**: Ajna protocol is based on the concept of *liquidity pool.* A pool is defined as a pair *collateral token/quote token* (e.g., ETH/DAI represents a pool where ETH is the collateral token and DAI is the quote token). It can be only a pool for a specific pair (e.g., there can be both ETH/DAI and DAI/ETH). When a user would lend, but there is still not a pool with a willing pair, he/she can create a new pool using the Ajna factory contract;
– **Quote token**: this token is always fungible, and it is supplied by lenders.
– **Collateral token**: this token can be fungible or non-fungible, and it is supplied by borrowers;
– **Price bucket**: to make easier the management of lending deposits, pools are split into *price buckets.* When a lender lends its quote tokens, he/she chooses the price buckets, i.e., chooses the price at which is willing to lend against the token.

Every Ajna pool has an index of prices, i.e., pools are split into price buckets. There are 7388 buckets in a pool, spaced 0.5% apart. Bucket 3232 represents the price of 1 (e.g. in an ETH/DAI pool, lenders that deposit their DAI in bucket 3232 are willing to lend 1 DAI for 1 ETH).

Category			Wardens		Bot Race	
			High	Medium	High	Medium
Business Logic	L0	Error in Business Logic Model	0	1	0	0
	L1	Freezing Active Position	3	0	0	0
	L2	Erroneous Accounting Business Implementation	1	2	0	0
	L3	Missing Business Logic Checks	4	3	0	0
	L4	Wrong Implementation	0	1	0	0
Solidity	S2	Precision Loss	0	2	0	0
	S3	Integer Overflow and Underflow	1	1	0	0
	S5	Arbitrarily Code Injection	1	0	0	0
	S6	Outdated Compiler Bug	0	1	0	0
	S7	Unsafe Casting	0	1	0	0
	S8	Using `transferFrom()` instead of `safeTransferFrom()` in ERC721 implementation	0	0	0	1
	S9	Using `_mint()` instead of `_safeMint()` in ERC721 implementation	0	0	0	1
EVM	E1	Gas Limit DoS	1	0	0	0
Blockchain	B1	Frontrunning	0	2	0	0
		Overall	11	14	0	2

Table 6: Ajna C4 Findings Overview.

C4 contest description There are 3 main contracts in the scope:

- **GrantFund**: this contract holds treasury, i.e., an amount of Ajna tokens that are used in protocol governance;
- **PositionManager**: a position of a lender in a given pool is represented using one or more Position NFTs. So, due to this, lenders can transfer their positions or stake them;
- **RewardsManager**: this contract rewards the Ajna lender who decides to stake his/her Ajna token using the RewardsManages contract itself.

C4 report description In Table 6 there is an overview of the results of human C4 contest participants, i.e., wardens, and the C4 bot race winner. Wardens reported many `Business Logic`. In particular, there are three *L1 - Freezing Active Position* issues with high severity.

C4 ID	C4 Title	Category
H-01	PositionManager's moveLiquidity can freeze funds by removing destination index even when the move was partial	L1 - Freezing Active Position
H-02	PositionManager's moveLiquidity can set wrong deposit time and permanently freeze LP funds moved	L1 - Freezing Active Position
H-03	Position NFT can be spammed with insignificant positions by anyone until rewards DoS	E1 - Gas Limit DoS
H-04	Delegation rewards are not counted toward granting fund	L2 - Erroneous Accounting Business Implementation
H-05	Incorrect calculation of the remaining updatedRewards leads to possible underflow error	S3 - Integer Overflow and Underflow
H-06	The lender could possibly lose unclaimed rewards in case a bucket goes bankrupt	L1 - Freezing Active Position
H-07	User can exponentially increase the value of their position through the memorializePositions function	L3 - Missing Business Logic Checks
H-08	Claiming accumulated rewards while the contract is underfunded can lead to a loss of rewards	L3 - Missing Business Logic Checks
H-09	User can avoid bankrupting by calling PositionManager.moveLiquidity where to index is bankrupted index	L3 - Missing Business Logic Checks
H-10	missing isEpochClaimed validation	L3 - Missing Business Logic Checks
H-11	RewardsManager fails to validate pool_ when updating exchange rates allowing rewards to be drained	S5 - Arbitrarily Code Injection

Table 7: Ajna highs.

C4 ID	C4 Title	Category
M-01	It is possible to steal the unallocated part of every delegation period budget	L3 - Missing Business Logic Checks
M-02	Delegate rewards system is unfair to delegates with less tokens and reduces decentralization	L0 - Error in Business Logic Model
M-03	_updateBucketExchangeRateAnd-CalculateRewards reward calculation accuracy loss	S2 - Precision Loss
M-04	Potential unfair distribution of Rewards due to MEV in updateBucketExchangeRatesAndClaim	B1 - Frontrunning
M-05	Calculating new rewards is susceptible to precision loss due to division before multiplication	S2 - Precision Loss
M-06	An Optimizer Bug in PositionManager.getPositionIndexesFiltered	S6 - Outdated Compiler Bug
M-07	Calling StandardFunding.screeningVote function and ExtraordinaryFunding.voteExtraordinary function when block.number equals respective start block and when block.number is bigger than respective start block can result in different available votes for same voter	L2 - Erroneous Accounting Business Implementation
M-08	The voting thresholds in Ajna's Extraordinary Funding Mechanism can be manipulated to execute proposals below the expected threshold	L3 - Missing Business Logic Checks
M-09	Adversary can prevent the creation of any extraordinary funding proposal by frontrunning proposeExtraordinary()	B1 - Frontrunning
M-10	Unsafe casting from uint256 to uint128 in RewardsManager	S7 - Unsafe Casting
M-11	StandardFunding.fundingVote should not allow users who didn't vote in screening stage to vote	L3 - Missing Business Logic Checks
M-12	Governance attack on Extraordinary Proposals	L2 - Erroneous Accounting Business Implementation
M-13	PositionManager & PermitERC721 Failure to comply with the EIP-4494	L4 - Wrong Implementation
M-14	PositionManager.moveLiquidity could revert due to underflow	S3 - Integer Overflow and Underflow

Table 8: Ajna mediums.

In Table 7 we assigned a category to every human high finding. As we can see, in this contest wardens reported three *L1 - Freezing Active Position.* As we said in Section 1.6, this issue is strongly related to the nature of DApp and the use of the *position* concept. In this case, we have a vulnerable contract, `PositionManager` that manages *positions*: when it does something wrong, it easily can incur in freezing positions.

In Table 8 there are Ajna medium findings of wardens. As we see above in human high findings, there are many *L3 - Missing Business Logic Checks* issues. Indeed, Ajna has a complex business model, which requires many important checks. Unfortunately, developers miss many of them, and this leads to several vulnerabilities.

In Table 9 there are Ajna bot race winner findings, both with corresponding categories. As we said in Section 1.17 and Section 1.18, this issue can be easily found by automatic tools. Indeed, in this contest, the bot race winner reported two valid medium issues. We underline that these issues are strongly related to the ERC721 standard, that is used in the Ajna contest to implement the *position* concept.

Analysis tools results We analyzed contracts from this C4 contest using Mythril, Slither, and Smartcheck. We report results in Table 10. In the Ajna contest, Slither managed to find three issues: two *S2 - Precision Loss* and one *S6 - Outdated Compiler Bug.* For the third one, we want to underline that Slither wasn't able to find the attack vector, but it only reports the usage of an outdated Solidity version. Anyway, we considered it as a valid finding, because its report could have helped Ajna developers.

2.2 Asymmetry

DApp description Asymmetry Finance is described in [15]. In Table 11 there is an overview of the 2023-03 contest. According to its whitepaper, "Asymmetry Finance is a protocol designed to bring a solution to the centralization of the staked Ether market".

Asymmetry uses the term *LSED* (Liquid Staked Ethereum Derivative) describing tokens based on staked ETH (like Lido's stETH and wstETH). Asymmetry aims to diversify ETH staking following several LSEDs. To do that, it creates its own LSED: safETH. Asymmetry's LSEDs are composed of stETH, rEHT, frxETH, sETH2, and ankrETH, and the protocol aims to increase the number of underlying LSEDs. When a new Asymmetry token is minted, the required amounts of each underlying LSED are acquired. Furthermore, through governance proposals, Asymmetry can change the underlying LSEDs, adding or removing them, and can change the weight with which every underlying LSED participates to Asymmetry token value.

User interaction with Asymmetry is simple: he/she can stake ETH and mint safETH. The quantity of minted safETH tokens is computed according to the underlying LSED. At this time, Asymmetry has a limitation on single transaction staking: from a minimum of 0.05 ETH to a maximum of 200 ETH.

C4 ID	C4 Title	Valid	Category
M-01	Use of transferFrom() rather than safeTransferFrom() for NFTs in will lead to the loss of NFTs	Y	S8 - Using `transferFrom()` instead of `safeTransferFrom()` in ERC721 implementation
M-02	_safeMint() should be used rather than _mint() wherever possible	Y	S9 - Using `_mint()` instead of `_safeMint()` in ERC721 implementation

Table 9: Ajna bot race winner findings.

Category			C4 report	Mythril	Slither	Smart-check
Business Logic	L0	Error in Business Logic Model	1	0	0	0
	L1	Freezing Active Position	3	0	0	0
	L2	Erroneous Accounting Business Implementation	3	0	0	0
	L3	Missing Business Logic Checks	7	0	0	0
	L4	Wrong Implementation	1	0	0	0
Solidity	S2	Precision Loss	2	0	2	0
	S3	Integer Overflow and Underflow	2	0	0	0
	S5	Arbitrarily Code Injection	1	0	0	0
	S6	Outdated Compiler Bug	1	0	1	0
	S7	Unsafe Casting	1	0	0	0
	S8	Using `transferFrom()` instead of `safeTransferFrom()` in ERC721 implementation	1	0	0	0
	S9	Using `_mint()` instead of `_safeMint()` in ERC721 implementation	1	0	0	0
EVM	E1	Gas Limit DoS	1	0	0	1
Blockchain	B1	Frontrunning	2	0	0	0
		Overall	27	0	3	1

Table 10: Ajna state-of-the-art automatic tools results.

Asymmetry				Derivative DApp	
	Contracts	Abstract contracts	Libraries	Interfaces	Overall
#	4	0	0	0	4
SLOCs	460	0	0	0	460
Pre-contest automatic tool analysis			Solhint		

Table 11: Asymmetry's 2023-03 contest.

C4 contest description There are 4 main contracts in the scope:

- **SafEth** Using this contract, a user can stake/unstake his/her funds. So, this contract has the core functions `mint()` and `burn()`
- **Reth, WstEth, and SfrxEth** These contracts contain methods to acquire rETH, wstETH, and sfrxETH, respectively.

We report a comprehensive image from the C4 contest in Figure 2.

C4 report description In Table 12 we report an overview of the 2023-04-Asymmetry contest final report. As we can see, this contest had many *Solidity* issues. It seems Asymmetry developers executed only the Solhint automatic tool on their smart contracts. Due to this, they didn't detect many *Solidity issues*.

In Table 13 we report Asymmetry C4 contest human high findings and the corresponding category. Many of these issues belong to *Business Logic* category, and so, are hard to detect. Humans report a *L0 - Error in Business Logic Model* high issue: *H06 - WstEth derivative assumes a 1=1 peg of stETH to ETH*. This is a good example of such vulnerability. A wrong assumption during the development of the business model could lead to create vulnerable projects.

Also in human medium findings (Table 14) there are two *L0 - Error in Business Logic Model* issues. In both cases, developers didn't take into account some aside scenarios, like the *de-peg* one, described in *M07 - In de-peg scenario, forcing full exit from every derivative & immediately re-entering can cause big losses for depositors.*

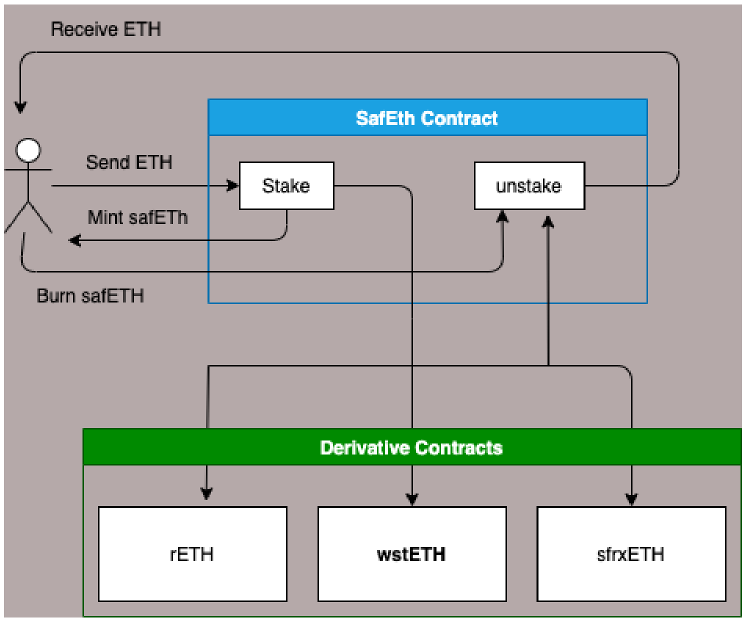

Fig. 2: Contract interactions in Asymmetry C4 contest

Category			Wardens		Bot Race	
			High	Medium	High	Medium
Business Logic	L0	Error in Business Logic Model	1	2	0	0
	L1	Freezing Active Position	0	1	0	0
	L2	Erroneous Accounting Business Implementation	2	1	0	0
	L3	Missing Business Logic Checks	1	2	0	0
Solidity	S2	Precision Loss	1	1	0	0
	S3	Integer Overflow and Underflow	1	0	0	0
	S11	Relying on External Source	1	2	0	0
	S12	Division by Zero DoS	0	1	0	0
	S13	Temporal DoS	0	1	0	0
Blockchain	B1	Frontrunning	1	1	0	0
		Overall	8	12	0	0

Table 12: Asymmetry C4 Findings Overview.

C4 ID	C4 Title	Category
H-01	An attacker can manipulate the preDepositvePrice to steal from other users	S2 - Precision Loss
H-02	A temporary issue shows in the staking functionality which leads to the users receiving less minted tokens	L2 - Erroneous Accounting Business Implementation
H-03	Users can fail to unstake and lose their deserved ETH because malfunctioning or untrusted derivative cannot be removed	S11 - Relying on External Source
H-04	Price of sfrxEth derivative is calculated incorrectly	L2 - Erroneous Accounting Business Implementation
H-05	Reth poolPrice calculation may overflow	S3 - Integer Overflow and Underflow
H-06	WstEth derivative assumes a 1=1 peg of stETH to ETH	L0 - Error in Business Logic Model
H-07	Reth.sol: Withdrawals are unreliable and depend on excess RocketDepositPool balance which can brick the whole protocol	L3 - Missing Business Logic Checks
H-08	Staking, unstaking and rebalanceToWeight can be sandwiched (Mainly rETH deposit)	B1 - Frontrunning

Table 13: Asymmetry highs.

Analysis tools results Using Mythril, Slither, and Smartcheck, we have analyzed the smart contracts of this contest. We show the results reported in Table 15. Slither found 4 *Solidity* issues: one *S2 - Precision Loss* and three *S3 - Integer Overflow and Underflow*. As we said above, especially for the *S3 - Integer Overflow and Underflow* ones, the pre-contest usage of the Slither tool could prevent humans from finding these vulnerabilities.

2.3 Caviar

DApp description Caviar is described in [16]. In Table 16 there is an overview of the 2023-04 contest. According to its documentation, "Caviar is an on-chain, gas-efficient automated market maker (AMM) protocol for trading non-fungible tokens (NFTs) with ERC20 tokens and ETH, supporting both whole and fractional amounts". Caviar permits users to deposit NFTs and associated assets inside *Liquidity Pools (LP)*. Then, using AMM, it is possible to trade NFTs for reserve assets.

There are two types of LPs. The first type is the **Shared Pools** where depositors can deposit NFT and reserve assets. Caviar uses the $x{\cdot}y = k$ invariant, like Uniswap. NFTs can classified as *floor*, *mid*, *rare*, and *grail* based on their desirability. Each of these classifications has its own shared pool.

C4 ID	C4 Title	Standardization proposal
M-01	Division before multiplication truncate minOut and incurs heavy precision loss and result in insufficient slippage protection	S2 - Precision Loss
M-02	sFrxEth may revert on redeeming non-zero amount	L1 - Freezing Active Position
M-03	Potential stake() DoS if sole safETH holder (ie: first depositor) unstakes totalSupply - 1	S12 - Division by Zero DoS
M-04	Lack of deadline for Uniswap AMM	B1 - Frontrunning
M-05	Missing derivative limit and deposit availability checks will revert the whole stake() function	S11 - Relying on External Source
M-06	DoS due to external call failure	S11 - Relying on External Source
M-07	In de-peg scenario, forcing full exit from every derivative & immediately re-entering can cause big losses for depositors	L0 - Error in Business Logic Model
M-08	Possible DoS on unstake()	S13 - Temporal DoS
M-09	Non-ideal rETH/WETH pool used pays unnecessary fees	L0 - Error in Business Logic Model
M-10	Stuck ether when use function stake with empty derivatives(derivativeCount = 0)	L3 - Missing Business Logic Checks
M-11	Residual ETH unreachable and unutilized in SafEth.sol	L2 - Erroneous Accounting Business Implementation
M-12	No slippage protection on stake() in SafEth.sol	L3 - Missing Business Logic Checks

Table 14: Asymmetry mediums.

The second type of LPs is the **Custom Pools**: these pools are similar to shared pools but can be created by a liquidity provider, who can set many customizable parameters, such as fee rates, flash loan support, etc.

Contest description The contest is mainly on `Custom Pools` implementation. The main contracts are:

- **Factory**: this contract only creates a new custom pool. Then, it creates an NFT that represents the ownership of that custom pool. This contract holds protocol fees;

Category			C4 report	Mythril	Slither	Smart-check
Business Logic	L0	Error in Business Logic Model	3	0	0	0
	L1	Freezing Active Position	1	0	0	0
	L2	Erroneous Accounting Business Implementation	3	0	0	0
	L3	Missing Business Logic Checks	3	0	0	0
Solidity	S2	Precision Loss	2	0	1	1
	S3	Integer Overflow and Underflow	1	0	0	0
	S11	Relying on External Source	3	0	3	0
	S12	Division by Zero DoS	1	0	0	0
	S13	Temporal DoS	1	0	0	0
Blockchain	B1	Frontrunning	2	0	0	0
		Overall	20	0	4	1

Table 15: Asymmetry state-of-the-art automatic tools results.

Caviar				DEX	
	Contracts	Abstract contracts	Libraries	Interfaces	Overall
#	4	0	0	1	5
SLOCs	730	0	0	11	741
Pre-contest automatic tool analysis			Slither		

Table 16: Caviar's 2023-04 contest.

– **PrivatePool**: this is where the core logic of custom pools is. An authorized user can use this contract to set all parameters of the pool. Traders can use this contract to perform their operation, i.e., they can buy, sell, and change NFTs for other NFTs in the pool;
– **EthRouter**: a user can use this contract to create a list of actions and perform them on various pools.

C4 report description In Table 17 there is an overview of the results of human C4 contest participants, i.e., wardens, and the C4 bot race winner. We can observe that all highs belong to *Solidity* category. In Table 18 we report more in detail the high issues found by humans.

We want to underline that in this contest, Slither didn't work correctly because of a bug, as reported by developers. We are pretty sure that without this problem, Slither was able to check these issues before the contest.

Category			Wardens		Bot Race	
			High	Medium	High	Medium
Business Logic	L0	Error in Business Logic Model	0	1	0	0
	L2	Erroneous Accounting Business Implementation	0	3	0	0
	L3	Missing Business Logic Checks	0	1	0	0
	L4	Wrong Implementation	0	1	0	0
Solidity	S1	Reentrancy	1	3	0	0
	S3	Integer Overflow and Underflow	0	2	0	0
	S5	Arbitrarily Code Injection	1	1	0	0
	S7	Unsafe Casting	1	0	0	0
	S11	Relying on External Source	0	3	0	0
	S14	Erroneous ERC721 Implementation	0	1	0	0
Blockchain	B1	Frontrunning	0	1	0	0
		Overall	3	17	0	0

Table 17: Caviar C4 Findings Overview.

In Table 19 there are human medium findings of the Caviar contest. Also there, we notice several *Solidity* issues that we explain with similar motivations. Such *Solidity* findings are a unique case among analyzed contests.

Analysis tools results We analyzed contracts from this C4 contest using Mythril, and Smartcheck. We reported the results in Table 20.

As we said above, there is a bug related to Slither that prevents it from running correctly. More info can be found here: `https://github.com/crytic/slither/issues/1737`. While we think Slither could find *Solidity* high issues, according to results in other contests, Mythril and Smartcheck didn't

C4 ID	C4 Title	Category
H-01	Royalty receiver can drain a private pool	S1 - Reentrancy
H-02	PrivatePool owner can steal all ERC20 and NFT from user via arbitrary execution	S5 - Arbitrarily Code Injection
H-03	Risk of silent overflow in reserves update	S7 - Unsafe Casting

Table 18: Caviar highs.

find any issues. So, in the Caviar contest, automatic tools have not found any vulnerabilities.

2.4 EigenLayer

EigenLayer				Derivative DApp	
	Contracts	Abstract contracts	Libraries	Interfaces	Overall
#	7	2	3	12	24
SLOCs	1023	42	231	97	1393
Pre-contest automatic tool analysis			Mythril		
			Slither		
			Solhint		

Table 21: EigenLayer's 2023-04 contest.

DApp description Eigenlayer is described in [17]. We report in Table 21 an overview of the 2023-04 contest.

According to its documentation, "[EigenLayer is] a restaking collective for Ethereum. EigenLayer is a set of smart contracts on Ethereum that allows consensus layer Ether (ETH) stakers to opt in to validate new software modules built on top of the Ethereum ecosystem".

EigenLayer aims to provide a new layer between Ethereum and stakers. We have already described Ethereum's proof of stake. EigenLayer gives stakers the possibility to restake their ETH on its ecosystem. In other words, this means that stakers give to "EigenLayer smart contracts the ability to impose additional slashing conditions on their staked ETH". Of course, this mechanism needs validators that can check transaction blocks and slash nodes. To do that, EigenLayer provides specific software to enable validators to participate in the EigenLayer ecosystem.

C4 ID	C4 Title	Category
M-01	The buy function's mechanism enables users to acquire flash loans at a cheaper fee rate.	S1 - Reentrancy
M-02	EthRouter can't perform multiple changes	S3 - Integer Overflow and Underflow
M-03	Flash loan fee is incorrect in Private Pool contract	L2 - Erroneous Accounting Business Implementation
M-04	changeFeeQuote will fail for low decimal ERC20 tokens	S3 - Integer Overflow and Underflow
M-05	EthRouter.sell, EthRouter.deposit, and EthRouter.change functions can be DOS'ed for some ERC721 tokens	S11 - Relying on External Source
M-06	Flashloan fee is not distributed to the factory	L0 - Error in Business Logic Model
M-07	Royalty recipients will not get fair share of royalties	L4 - Wrong Implementation
M-08	Loss of funds for traders due to accounting error in royalty calculations	L3 - Missing Business Logic Checks
M-09	Malicious royalty recipient can steal excess eth from buy orders	S1 - Reentrancy
M-10	Incorrect protocol fee is taken when changing NFTs	L2 - Erroneous Accounting Business Implementation
M-11	Factory.create: Predictability of pool address creates multiple issues.	B1 - Frontrunning
M-12	Prohibition to create private pools with the factory NFT	S1 - Reentrancy
M-13	Transaction revert if the baseToken does not support 0 value transfer when charging changeFee	S11 - Relying on External Source
M-14	The royaltyRecipient could not be prepare to receive ether, making the sell to fail	S11 - Relying on External Source
M-15	Pool tokens can be stolen via PrivatePool.flashLoan function from previous owner	S5 - Arbitrarily Code Injection
M-16	PrivatePool.flashLoan() takes fee from the wrong address	L2 - Erroneous Accounting Business Implementation
M-17	The tokenURI method does not check if the NFT has been minted and returns data for the contract that may be a fake NFT	S14 - Erroneous ERC721 Implementation

Table 19: Caviar mediums.

Category			C4 report	Mythril	Slither	Smart-check
Business Logic	L0	Error in Business Logic Model	1	0	0	0
	L2	Erroneous Accounting Business Implementation	3	0	0	0
	L3	Missing Business Logic Checks	1	0	0	0
	L4	Wrong Implementation	1	0	0	0
Solidity	S1	Reentrancy	4	0	0	0
	S3	Integer Overflow and Underflow	2	0	0	0
	S5	Arbitrarily Code Injection	2	0	0	0
	S7	Unsafe Casting	1	0	0	0
	S11	Relying on External Source	3	0	0	0
	S14	Erroneous ERC721 Implementation	1	0	0	0
Blockchain	B1	Frontrunning	1	0	0	0
		Overall	20	0	0	0

Table 20: Caviar state-of-the-art automatic tools results.

EigenLayer also provides an open market mechanism that enables validators to choose the modules they want to opt in. Finally, EigenLayer provides several ways to restake ETHs. It can be done in ETH natively way, too. Furthermore, the protocol allows to use of LSTs (Liquid Staking Tokens) obtained via staking pool protocols. This mechanism is quite similar to the most famous one: Lido.

Contest description There are 7 contracts[11].

– **StrategyManager** This contract tracks stakers' deposit using a so-called **shares**, that represent a staker holding. When someone wants to deposit ERC20 tokens can use this contract that will transfer those tokens to a user-specified Strategy contract. This contract also provides withdrawn functionalities.

[11] Technical specifies are in `https://github.com/code-423n4/2023-04-eigenlayer/blob/main/docs/EigenLayer-tech-spec.md`

Category			Wardens		Bot Race	
			High	Medium	High	Medium
Business Logic	L0	Error in Business Logic Model	0	1	0	0
	L2	Erroneous Accounting Business Implementation	1	0	0	0
	L3	Missing Business Logic Checks	1	0	0	0
Solidity	S11	Relying on External Source	0	1	0	0
		Overall	2	2	0	0

Table 22: EigenLayer C4 Findings Overview.

C4 ID	C4 Title	Category
H-01	Slot and block number proofs not required for verification of withdrawal (multiple withdrawals possible)	L3 - Missing Business Logic Checks
H-02	It is impossible to slash queued withdrawals that contain a malicious strategy due to a misplacement of the ++i increment	L2 - Erroneous Accounting Business Implementation

Table 23: EigenLayer highs.

- **StrategyBase** This contract represents the base for every Strategy contract. Each Strategy has to manage a single underlying ERC20.
- **EigenPodManager** This contract allows users to stake ETH. Furthermore, it coordinates deposits and withdrawals to and from StrategyManager.
- **EigenPod** Each staker can deploy an EigenPod, that allows him to stake ETHs on Ethereum and restake them on EigenLayer.
- **DelayedWithdrawalRouter** This contract is a control over withdrawals of ETH from EigenPods
- **Pausable** This contract is used to extend other contracts and makes them pausable, i.e., stops their functionalities.
- **PauserRegistry** This contract holds pauser and unpauser roles.

C4 report description In Table 22 there is an overview of the results of human C4 contest participants, i.e., wardens, and the C4 bot race winner. As we can see,

C4 ID	C4 Title	Category
M-01	A staker with verified over-commitment can potentially bypass slashing completely	L0 - Error in Business Logic Model
M-02	A malicious strategy can permanently DoS all currently pending withdrawals that contain it	S11 - Relying on External Source

Table 24: EigenLayer mediums.

humans found only 4 issues. Before the contest, EigenLayer developers executed three automatic tools: Mythril, Slither, and Solhint. As a consequence, humans found only one *Solidity* issue.

In Table 23 we report human high findings and in Table 24 there are human medium findings. We can see that humans reported a *L0 - Error in Business Logic Model* issue that causes the staker to bypass the slashing mechanism. The unique *Solidity* issue is an instance of the *Relying on External Source* vulnerability. In Table 25 we report the bot race winner findings. In this contest, the winner bot found only one medium issue. However, it was not considered valid by the judge.

C4 ID	C4 Title	Valid	Category
M-01	Contracts are vulnerable to fee-on-transfer-token-related accounting issues	N	

Table 25: EigenLayer automatic findings.

Analysis tools results In Table 26 we have reported the result of automatic tools analysis, performed as usual, using Mythril, Slither, and Smartcheck. As we said before, humans found only one *Solidity* issue. This issue was found by Slither, too. Developers didn't check Slither's report or they didn't consider its report implications. On the other hand, human experts manage to write a proof of concept of that vulnerability.

Category			C4 report	Mythril	Slither	Smartcheck
Business Logic	L0	Error in Business Logic Model	1	0	0	0
	L2	Erroneous Accounting Business Implementation	1	0	0	0
	L3	Missing Business Logic Checks	1	0	0	0
Solidity	S11	Relying on External Source	1	0	1	0
		Overall	4	0	1	0

Table 26: EigenLayer state-of-the-art automatic tools results.

2.5 ENS

ENS				Service DApp	
	Contracts	Abstract contracts	Libraries	Interfaces	Overall
#	9	0	9	0	18
SLOCs	1025	0	997	0	2022
Pre-contest automatic tool analysis					

Table 27: ENS's 2023-04 contest.

DApp description ENS is described in [18]. In Table 27 there is an overview of the 2023-04 contest. According to its documentation, "The Ethereum Name Service (ENS) is a distributed, open, and extensible naming system based on the Ethereum blockchain".

This DApp provides functionalities to map human-readable names to machine-readable identifiers (e.g., Ethereum addresses). It is inspired by DNS (Domain Name System), which is the name system used on the internet to associate a *domain name* to various information. ENS uses a system of hierarchical domains: there is a top-level domain, owned by a so-called `registrars` contracts. Registrars allow everybody, according to specific rules, to allocate a *subdomain*. Once someone obtains ownership of a subdomain, he/she can create and configure its subdomains, and so on.

The ENS Architecture is a complicated environment, but essentially it has two principal components: the `registry` and the `resolvers`. The ENS `registry` maintains a list of all domains and subdomains with their important information: the owner, the resolver, and the time-to-live of records under the domain. `Resolvers` translating names into addresses. When a user needs to resolve a name, ask to ENS Registry for the list of resolvers that solve the address domain. Then, ask that resolver to translate the name.

Contest description There are 3 main contracts in the scope:

- **DNSRegistrar**: this is the contract that allows to register DNS names (like .com, .org, .net, etc.) on **registry** using DNSSEC Oracle;
- **DNSClaimChecker**: this contract verifies DNS name claims;
- **OffchainDNSResolver**: this contract is a resolver contract that handles gasless DNS name resolution.

Category			Wardens		Bot Race	
			High	Medium	High	Medium
Business Logic	L2	Erroneous Accounting Business Implementation	0	3	0	0
	L3	Missing Business Logic Checks	0	1	0	0
	L4	Wrong Implementation	0	2	0	0
Solidity	S7	Unsafe Casting	0	1	0	0
		Overall	0	7	0	0

Table 28: ENS C4 Findings Overview.

C4 report description In Table 28 there is an overview of the results of human C4 contest participants, i.e., wardens, and the C4 bot race winner. We just reported Table 29 of human medium findings, because in this contest there weren't high findings. Moreover, the bot race winner didn't report any high or medium issues. As we can see from these tables, almost all issues belong to the *Business Logic* category. In particular, there are two *Wrong Implementation* issues. In these cases, developers implement wrongly the business model, introducing these vulnerabilities.

Analysis tools results As in the previous contests, we used Mythril, Slither, and Smartcheck to analyze ENS. We have reported the results in Table 30. As we said above, almost all issues belong to the *Business Logic* category. Indeed, automatic tools were not able to find them.

C4 ID	C4 Title	Category
M-01	HexUtils.hexStringToBytes32() and HexUtils.hexToAddress() may return incorrect results	L2 - Erroneous Accounting Business Implementation
M-02	Invalid addresses will be accepted as resolvers, possibly bricking assets	S7 - Unsafe Casting
M-03	Offchain name resolution would fail despite the located DNS resolver being fully functional	L4 - Wrong Implementation
M-04	Incorrect implementation of RecordParser.readKeyValue()	L2 - Erroneous Accounting Business Implementation
M-05	Unintentionally register a non-relevant DNS name owner	L3 - Missing Business Logic Checks
M-06	validateSignature(...) in EllipticCurve mixes up Jacobian and projective coordinates	L2 - Erroneous Accounting Business Implementation
M-07	Missing recursive calls handling in OffchainDNSResolver CCIP-aware contract	L4 - Wrong Implementation

Table 29: ENS mediums.

2.6 Frankencoin

Frankencoin				Stablecoin	
	Contracts	Abstract contracts	Libraries	Interfaces	Overall
#	8	2	0	0	10
SLOCs	820	129	0	0	949
Pre-contest automatic tool analysis			Slither		

Table 31: Frankencoin's 2023-04 contest.

DApp description Frankencoin is described in [19]. In Table 31 there is an overview of the 2023-04 contest. According to its documentation, "The Frankencoin is a collateralized stablecoin that is intended to track the value of the Swiss Franc. Its governance is decentralized, with anyone being able to propose new minting mechanisms and anyone who owns more than 2% of the voting power has the power to veto proposals."

Frankencoin is an implementation of a stablecoin pegged 1=1 with the Swiss Franc. The main goal of this protocol is to create a stablecoin that doesn't rely

Category			C4 report	Mythril	Slither	Smartcheck
Business Logic	L2	Erroneous Accounting Business Implementation	3	0	0	0
	L3	Missing Business Logic Checks	1	0	0	0
	L4	Wrong Implementation	2	0	0	0
Solidity	S7	Unsafe Casting	1	0	0	0
		Overall	7	0	0	0

Table 30: ENS state-of-the-art automatic tools results.

on an external oracle. In this way, Frankencoin's value doesn't rely on an external source that can be manipulated and attacked.

Contest description The focus of this contest is the core contracts of Frankencoin architecture, that allow Frankencoin tokens (ZCHF) minting, the purchase of reserve pool shares, and the conversion between XCHF (i.e., CryptoFranc) and ZCHF. In particular:

– **Position** represents the position of a user and holds his/her collateral allowing to mint new Frankencoins;
– **MintingHub** creates and challenge positions;
– **Frankencoin** the Frankencoin ERC20 token;
– **StablecoinBridge** a contract that allows to swap Frankencoin with other stablecoins.

C4 report description In Table 32 there is an overview of the results of wardens and the C4 bot race winner. In this contest, wardens found many vulnerabilities: 6 high and 15 medium human findings. There are many *L0 - Error in Business Logic Model* issues. We believe this was caused by the nature of this DApp. Frankencoin aims to develop stablecoin pegged to the Swiss Franc, as we described above. This purpose is very complex. In the last years, several stablecoins that appeared sturdy, are collapsed [20].

In Table 33 and Table 34, we report Frankencoin human findings. In addition to the already mentioned *L0 - Error in Business Logic Model* issues, humans found 6 *Blockchain* issues. In particular, they reported 5 *Frontrunning* vulnerabilities. As we said in Section 1.1, Frontrunning is an emergent problem. Because Frankencoin implements decentralized governance and aims to stabilize its value without the use of external sources, it can be more vulnerable than other DApps to frontrunning attacks.

Analysis tools results In Table 35 we can see that Mythril, Slither, and Smartcheck didn't find any vulnerability. However, in this contest, humans found

Category			Wardens		Bot Race	
			High	Medium	High	Medium
Business Logic	L0	Error in Business Logic Model	2	4	0	0
	L1	Freezing Active Position	1	1	0	0
	L2	Erroneous Accounting Business Implementation	0	3	0	0
	L3	Missing Business Logic Checks	0	1	0	0
Solidity	S2	Precision Loss	0	1	0	0
	S3	Integer Overflow and Underflow	1	0	0	0
	S11	Relying on External Source	0	1	0	0
Blockchain	B1	Frontrunning	2	3	0	0
	B2	Erroneous Blockchain Assumption	0	1	0	0
		Overall	6	15	0	0

Table 32: Frankencoin C4 Findings Overview.

just 3 *Solidity* issues. We didn't expect that automatic tools would be able to find *Business Logic* or *Blockchain* issues.

2.7 Juicebox

Juicebox				Service DApp	
	Contracts	Abstract contracts	Libraries	Interfaces	Overall
#	1	0	0	0	1
SLOCs	160	0	0	0	160
Pre-contest automatic tool analysis			Solhint		

Table 36: Juicebox's 2023-05 contest.

DApp description Juicebox is described in [21]. In Table 36 there is an overview of the 2023-05 contest. According to its documentation, "The Juicebox protocol is a programmable treasury. Projects can use it to configure how its tokens should be minted when it receives funds, and under what conditions those funds can be distributed to preprogrammed addresses or reclaimed by its community".

C4 ID	C4 Title	Category
H-01	Challenges can be frontrun with deleveraging to cause losses for challengers	B1 - Frontrunning
H-02	Double-entrypoint collateral token allows position owner to withdraw underlying collateral without repaying ZCHF	L0 - Error in Business Logic Model
H-03	When the challenge is successful, the user can send tokens to the position to avoid the position's cooldown period being extended	B1 - Frontrunning
H-04	Transfer position ownership to addr(0) to DoS end() challenge	L1 - Freezing Active Position
H-05	Position owners can deny liquidations	S3 - Integer Overflow and Underflow
H-06	CHALLENGER_REWARD can be used to drain reserves and free mint	L0 - Error in Business Logic Model

Table 33: Frankencoin highs.

When a user creates a new treasury using Juicebox protocol, he/she mints an NFT that represents his/her ownership and allows configuring treasury parameters. The protocol provides an architecture that can be easily extended by users to create their custom treasury contracts.

Contest description In the contest's scope, there is only one contract, JBXBuybackDelegate, which provides functionality described in **Buyback Delegate** description[12]. When `pay()` function is called, it sends an amount of project token to the contributor computed according to the number of ethers the contributor is sending to the treasure. To do so, `pay()` uses Buyback Delegate, which can select the best choice between minting new project tokens or using Uniswap and swapping ethers and project tokens.

C4 report description In Table 37 we report the results of wardens and the bot race winner of the 2023-05-Juicebox contest. As we can see, there are only *Business Logic* findings, all of them with medium severity. In a contest composed of one smart contract, developers analyzed the code, both through visual

[12] `https://docs.juicebox.money/dev/extensions/juice-buyback/`

C4 ID	C4 Title	Category
M-01	Function restructureCapTable() in Equity.sol not functioning as expected	L2 - Erroneous Accounting Business Implementation
M-02	POSITION LIMIT COULD BE FULLY REDUCED TO ZERO BY CLONES	L2 - Erroneous Accounting Business Implementation
M-03	Manipulation of total share amount might cause future depositors to lose their assets	L3 - Missing Business Logic Checks
M-04	anchorTime() will not work properly on Optimism due to use of block.number	B1 - Frontrunning
M-05	Owner of Denied Position is not able to withdraw collateral until expiry	L1 - Freezing Active Position
M-06	Challengers and bidders can collude together to restrict the minting of position owner	L0 - Error in Business Logic Model
M-07	Need alternative ways for fund transfer in end() to prevent DoS	S11 - Relying on External Source
M-08	initializeClone() price calculation should round up	S2 - Precision Loss
M-09	Unable to adjust position in some cases	L2 - Erroneous Accounting Business Implementation
M-10	No slippage control when minting and redeeming FPS	B1 - Frontrunning
M-11	Later challengers can bid on the previous challenge to extend the expiration time of the previous challenge, so that their own challenge can succeed before the previous challenge and get challenge rewards	L0 - Error in Business Logic Model
M-12	Auctions fail to account for network and market conditions	L0 - Error in Business Logic Model
M-13	Can't pause or remove a minter	L0 - Error in Business Logic Model
M-14	Re-org attack in factory	B2 - Erroneous Blockchain Assumption
M-15	notifyLoss can be frontrun by redeem	B1 - Frontrunning

Table 34: Frankencoin mediums.

Category			C4 report	Mythril	Slither	Smartcheck
Business Logic	L0	Error in Business Logic Model	6	0	0	0
	L1	Freezing Active Position	2	0	0	0
	L2	Erroneous Accounting Business Implementation	3	0	0	0
	L3	Missing Business Logic Checks	1	0	0	0
Solidity	S2	Precision Loss	1	0	0	0
	S3	Integer Overflow and Underflow	1	0	0	0
	S11	Relying on External Source	1	0	0	0
Blockchain	B1	Frontrunning	5	0	0	0
	B2	Erroneous Blockchain Assumption	1	0	0	0
		Overall	21	0	0	0

Table 35: Frankencoin state-of-the-art automatic tools results.

Category			Wardens		Bot Race	
			High	Medium	High	Medium
Business Logic	L1	Freezing Active Position	0	1	0	0
	L2	Erroneous Accounting Business Implementation	0	1	0	0
	L4	Wrong Implementation	0	1	0	0
		Overall	0	3	0	0

Table 37: Juicebox C4 Findings Overview.

inspection and automatic tools (they used Solhint analysis before the contest). In Table 38 we report the details of findings of wardens. As we see in *M02 - Low Liquidity in Uniswap V3 Pool Can Lead to ETH Lockup in JBXBuybackDelegate Contract*, certain conditions of external sources, in this case *Uniswap*, could lead to *L1 - Freezing Active Position* issue.

In Table 39 there are Juicebox bot race winner findings with corresponding category. We have considered all invalid. Even if the Code4rena team shared them on the Juicebox contest webpage, it seems they also don't consider these issues valid, because they didn't write them in the final report[13]. For example, the bot winner reported *M01 - Fee-on-transfer/rebasing tokens will have problems when swapping.* This issue can be caused by some tokens that can be not supported by Uniswap. However, developers don't use those tokens and wrote this information in the contest *Scoping detail.* This is an interesting example of how a bot should know some characteristics of the DApp environment to analyze it.

[13] `https://code4rena.com/reports/2023-05-juicebox`

C4 ID	C4 Title	Category
M-01	Delegate architecture forces users to set zero slippage	L2 - Erroneous Accounting Business Implementation
M-02	Low Liquidity in Uniswap V3 Pool Can Lead to ETH Lockup in JBXBuyback-Delegate Contract	L1 - Freezing Active Position
M-03	Funding cycles that use JBXBuyback-Delegate as a redeem data source (or any derived class) will slash all re-deemable tokens	L4 - Wrong Implementation

Table 38: Juicebox mediums.

C4 ID	C4 Title	Valid	Category
M-01	Fee-on-transfer/rebasing tokens will have problems when swapping	N	
M-02	Unsafe use of trans-fer()/transferFrom() with IERC20	N	
M-03	Return values of trans-fer()/transferFrom() not checked	N	

Table 39: Juicebox automatic findings.

Analysis tools results In Table 40 we report the results of Mythril, Slither, and Smartcheck analysis. As we said above, wardens found just three *Business Logic* issues. For these issues, no automatic tool has detectors to find them.

2.8 Livepeer

Livepeer				Social DApp	
	Contracts	Abstract contracts	Libraries	Interfaces	Overall
#	4	1	2	3	10
SLOCs	1278	124	98	105	1605
Pre-contest automatic tool analysis				Slither	
				Solhint	

Table 41: Livepeer's 2023-08 contest.

Category			C4 report	Mythril	Slither	Smartcheck
Business Logic	L1	Freezing Active Position	1	0	0	0
	L2	Erroneous Accounting Business Implementation	1	0	0	0
	L4	Wrong Implementation	1	0	0	0
		Overall	3	0	0	0

Table 40: Juicebox state-of-the-art automatic tools results.

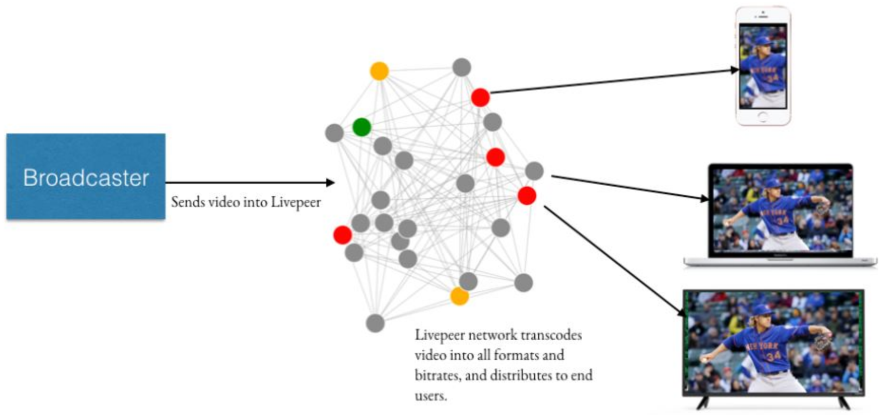
Broadcaster
Sends video into Livepeer
Livepeer network transcodes video into all formats and bitrates, and distributes to end users.

Fig. 3: Livepeer original idea from Livepeer Whitepaper.

DApp description Livepeer is described in [22][23]. In Table 41 there is an overview of the 2023-08 contest. According to its documentation, "Livepeer is a video infrastructure network for live and on-demand streaming. Designed to give developers the freedom to innovate, creators autonomy from platforms, and viewers a choice in their experience."

In Figure 3 there is a visual description of the scope of Livepeer. This DApp aims to provide a video live-streaming service built on top of Ethereum. Several entities work to create and power the Livepeer network. The *Broadcaster* sends video to Livepeer, asking the network to hold and send video streams to watchers. *Orchestrators* work as proxies. They receive videos from broadcasters and send them to various *transcoders*, that encode the video into various formats to reach every supported device. After this, they send the encoded video stream back to *orchestrators*, which deliver the video to *consumers*. To make this network work, there are two types of incentive: the first is the value provided by *broadcaster*, that have to send ETH to the network when he/she asks to broadcast a video. This incentive is distributed among *orchestrators* and *transcoders*. Furthermore, when the *orchestrators* work well, they increase the likelihood of receiving future work, and so, future ETH from *broadcaster*. Finally, to avoid wrong or malicious behavior of *orchestrators*, they have to stake an amount of Livepeer Token, and so, they could occur in penalties.

Contest description This contest involves contracts that aim to build an on-chain community treasury as a percentage of the protocol rewards.

– **BondingManager**: this contract manages protocol staking and rewards. It has been updated with new functionalities to support state checkpointing and treasury rewards minting;
– **BondingVotes**: this contract holds state checkpointing;
– **Treasury**: this contract holds funds of treasury and executes proposals;
– **LivepeerGovernor** OpenZeppelin Governor implementation of Livepeer.

Category			Wardens		Bot Race	
			High	Medium	High	Medium
Business Logic	L2	Erroneous Accounting Business Implementation	0	1	0	0
	L3	Missing Business Logic Checks	1	0	0	0
	L4	Wrong Implementation	0	2	0	0
Solidity	S3	Integer Overflow and Underflow	1	0	0	0
		Overall	2	3	0	0

Table 42: Livepeer C4 Findings Overview.

C4 report description In Table 42 there is an overview of the results of human C4 contest participants, i.e., wardens, and the C4 bot race winner. While the bot race winner didn't find any high or medium issues, wardens report 4 *Business Logic* and 1 *Solidity* problems. Also in this contest, it seems that participants in bot races can't find issues that belong to the *Business Logic* category.

In Table 43 and in Table 44 we report high and medium findings of wardens, both with corresponding category. As we can see, among high findings there is a *S3 - Integer Overflow and Underflow* issue: *H01 - Underflow in updateTranscoderWithFees can cause corrupted data and loss of winning tickets.* Let's see it in detail[14]. We consider this problem as *S3 - Integer Overflow and*

[14] `https://code4rena.com/reports/2023-08-livepeer#h-01-underflow-in-updatetranscoderwithfees-can-cause-corrupted-data-and-loss-of-winning-tickets`

C4 ID	C4 Title	Category
H-01	Underflow in updateTranscoderWith-Fees can cause corrupted data and loss of winning tickets	S3 - Integer Overflow and Underflow
H-02	By delegating to a non-transcoder, a delegator can reduce the tally of some-one else's vote choice without first granting them any voting power	L3 - Missing Business Logic Checks

Table 43: Livepeer highs.

C4 ID	C4 Title	Category
M-01	The logic in _handleVoteOverride to determine if an account is the transcoder is not consistent with the logic in the BondManager.sol	L4 - Wrong Implementation
M-02	Fully slashed transcoder can vote with 0 weight messing up the voting calcu-lations	L2 - Erroneous Accounting Business Implementation
M-03	withdrawFees does not update check-point	L4 - Wrong Implementation

Table 44: Livepeer mediums.

Underflow because it causes an underflow, with the consequences described in Section 1.12. This issue is due to the wrong usage of math libraries. Developers used *MathUtils* in one part of the code, and *PreciseMathUtils* in another part: these two libraries use different amounts of digits in their computation. So, this issue could be considered as a *S2 - Precision Loss*. However, we didn't think so: even if there is a precision loss, it should have no consequences to security. Developers had already noticed the precision loss problem, and for this reason, they introduced *MathUtils* and *PreciseMathUtils* libraries. Instead, the reported issue is a consequence of two different precision losses which causes the *undeflow* problem.

Analysis tools results In Table 45 there are the results of our analysis on the contest, performed using Mythril, Slither, and Smartcheck. Above, we have described the *Solidity* issue found by wardens during the contest. Among used automatic tools, Mythril is the only which has a detector to intercept the *Integer Overflow and Underflow* issue. However, Mythril can detect only when a variable underflows, and not when such underflow causes the revert of transaction. To make the exposition clear, let's use an example. We define an `uint8` variable with an initial value of 10, and then we perform a subtraction:

Category			C4 report	Mythril	Slither	Smartcheck
Business Logic	L2	Erroneous Accounting Business Implementation	1	0	0	0
	L3	Missing Business Logic Checks	1	0	0	0
	L4	Wrong Implementation	2	0	0	0
Solidity	S3	Integer Overflow and Underflow	1	0	0	0
		Overall	5	0	0	0

Table 45: Livepeer state-of-the-art automatic tools results.

```
uint8 x = 10;
x = x-50; // x now is 255;
```

Assuming that we are using a Solidity version less than 0.8.0 (see Section 1.12) or the code is inside an `unchecked` statement, the new value of x is

$$x = (10 - 50)mod(2^8) = 216$$

So, if the underflow operation happens, we have a mathematically wrong result. Indeed, if we compute the inverse operation

$$216 + 50 = 10$$

it will be valid only if we apply the $mod(2^8)$ operator (i.e., if the operation overflow). In this case, Mythril can intercept the issue, because it notices that the operation didn't return the right value unless we consider the $mod(2^k)$ operator. In the issue reported by wardens, the wrong value is never assigned to the variable, but the transaction reverts before. Since Mythril detectors were written before the introduction of the OpenZeppelin SafeMath library by default, developers of that tool didn't write code to intercept when a transaction reverts because an overflow/underflow issue happens.

2.9 Llama

Llama				Service DApp	
	Contracts	Abstract contracts	Libraries	Interfaces	Overall
#	11	4	2	3	20
SLOCs	1572	264	176	35	2047
Pre-contest automatic tool analysis			Slither		

Table 46: Llama's 2023-06 contest.

DApp description Llama is described in [24]. In Table 46 there is an overview of the 2023-06 contest. According to its documentation, "Llama is an on-chain governance and access control framework for smart contracts. Using Llama, teams can deploy fully independent instances that define granular roles and permissions for executing transactions, known as *actions*".

Llama provides a governance framework to make life easier for developers. Using Llama, who develops a new DApp can create a governance policy that relies on Llama architecture. The Llama environment is composed of 4 main concepts:

- **Policies**: they are non-transferable NFTs and define the Llama instance's roles and permissions;
- **Actions**: they are proposals of transactions initiated by *policyholders*;
- **Strategies**: they are contracts with rules for how to create and execute *actions*;
- **Accounts**: they are contracts that manage on-chain assets.

Contest description The contest involves the main contracts of Llama architecture

- **LlamaCore**: this contract manages *action* process. It defines *strategies* of *action* and, if it is needed, *guard* functionalities in order to check the correct execution of *actions*;
- **LlamaPolicy**: this contracts is an ERC721 contract that holds *policy* NFTs.
- **LlamaExeutor**: this contract executes *actions*, according to their *approval periods*, during that policyholders can approve *actions.*

Category			Wardens		Bot Race	
			High	Medium	High	Medium
Business Logic	L2	Erroneous Accounting Business Implementation	1	0	0	0
	L3	Missing Business Logic Checks	1	0	0	0
	L4	Wrong Implementation	0	1	0	0
Solidity	S9	Using `_mint()` instead of `_safeMint()` in ERC721 implementation	0	0	0	1
EVM	E1	Gas Limit DoS	0	1	0	0
Blockchain	B1	Frontrunning	0	1	0	0
		Overall	2	3	0	1

Table 47: Llama C4 Findings Overview.

C4 ID	C4 Title	Category
H-01	In LlamaRelativeQuorum, the governance result might be incorrect as it counts the wrong approval/disapproval	L2 - Erroneous Accounting Business Implementation
H-02	Anyone can change approval/disapproval threshold for any action using LlamaRelativeQuorum strategy	L3 - Missing Business Logic Checks

Table 48: Llama highs.

C4 ID	C4 Title	Category
M-01	It is not possible to execute actions that require ETH (or other protocol token)	L4 - Wrong Implementation
M-02	User with disapproval role can gas grief the action executor	B1 - Frontrunning
M-03	LlamaPolicy could be DOS by creating large amount of actions	E1 - Gas Limit DoS

Table 49: Llama mediums.

C4 report description In Table 47 we report an overview of the findings of wardens and the bot race winner. As we can see, there is at least one issue per category. In particular, the bot race winner detected one *S9 - Using `_mint()` instead of `_safeMint()` in ERC721 implementation* issue, that we describe in Section 1.18. In Table 48 and Table 49, there are high and medium findings from wardens, respectively. Wardens report three *Business Logic* issues, one *B1 - Frontrunning* and one *E1 - Gas Limit DoS*. In particular, the last one was found in the `LlamaPolicy contract`: wardens showed that an attacker could create an arbitrary number of actions, causing the DoS of the protocol's policy.

C4 ID	C4 Title	Valid	Category
M-01	Some tokens may revert when zero value transfers are made	N	
M-02	_safeMint() should be used rather than_mint() wherever possible	Y	S9 - Using `_mint()` instead of `_safeMint()` in ERC721 implementation

Table 50: Llama automatic findings.

In Table 50 we report the bot race winner findings. It found one *S9 - Using _mint() instead of _safeMint() in ERC721 implementation*. Furthermore, it reports a *M01 - Some tokens may revert when zero value transfers are made* that we consider not valid. Indeed, this finding is valid only in the case DApp uses a certain token, like *LEND*, that reverts when a zero transfer is made. Llama doesn't use these tokens. Furthermore, the reported code is not used in a loop or in a batch of transferring, so even if it reverts, has no impact on the protocol.

Category			C4 report	Mythril	Slither	Smart-check
Business Logic	L2	Erroneous Accounting Business Implementation	1	0	0	0
	L3	Missing Business Logic Checks	1	0	0	0
	L4	Wrong Implementation	1	0	0	0
Solidity	S9	Using `_mint()` instead of `_safeMint()` in ERC721 implementation	1	0	0	0
EVM	E1	Gas Limit DoS	1	0	0	0
Blockchain	B1	Frontrunning	1	0	0	0
		Overall	6	0	0	0

Table 51: Llama state-of-the-art automatic tools results.

Analysis tools results In Table 51 we report the results of automatic tools analysis. Tools didn't find any issue. Despite Smartcheck managing to find the *E1 - Gas Limit DoS* issue in the 2023-05-Ajna contest, in this case, it didn't report anything.

2.10 Shell

Shell				Service DApp	
	Contracts	Abstract contracts	Libraries	Interfaces	Overall
#	1	0	0	0	1
SLOCs	460	0	0	0	460
Pre-contest automatic tool analysis					

Table 52: Shell's 2023-08 contest.

DApp description Shell is described in [25]. In Table 52 there is an overview of the 2023-08 contest. According to its documentation, the original purpose in 2020 of the protocol was "to create an internet monetary system: a borderless, programmable medium of exchange accessible to all".

After two years, in 2022, the Shell Team published a new whitepaper [26], where they defined the so-called *Ocean*. The *Ocean* is the main concept of the Shell Protocol. From the whitepaper: "the objective of the Ocean is to create a platform that can compose any type of primitive: AMMs, lending pools, algorithmic stablecoins, NFT markets, and even future primitives yet to be invented. It should also support popular token standards: ERC-20, ERC721, and ERC-1155".

In standard DeFi architecture, there are many instruments that we can call primitives, i.e., objects that are used in many different contexts, but usually in similar ways. Examples of primitives are AMM, lending pools, stablecoins, NFT, and so on. In general, the composition of these primitives can be onerous and sometimes it may have to reinvent the wheel. Furthermore, sometimes these compositions have to be created one transaction at a time, and this can be gas-expensive and error-prone. The *Ocean* aims to provide an infrastructure that makes managing and composing these primitives easier.

To do so, the *Ocean* implements a single contract ledger, which holds all tokens, generalizes accounting logic, provides an interface that supports any kind of primitive, and, during primitive composition, tracks intermediate balances in memory, to save gas, and only at the end saves them in storage.

Contest description The contest involves only one contract: **EvolvingProteus**. This contract represents the primitive of AMM. i.e., a liquidity pool and its evolution over time. The user who defines a new pool instance on the Ocean has two possibilities: **Dutch auction** to create a liquidity concentration that holds the user's funds and can sell them over a specified time duration, and **Dynamic pools** that are more similar to AMM, where the liquidity concentration evolves dynamically over time according to external variables.

Category			Wardens		Bot Race	
			High	Medium	High	Medium
Business Logic	L3	Missing Business Logic Checks	1	0	0	0
		Overall	1	0	0	0

Table 53: Shell C4 Findings Overview.

C4 ID	C4 Title	Category
H-01	Lack of Balance Validation	L3 - Missing Business Logic Checks

Table 54: Shell highs.

C4 report description In Table 53 there is an overview of findings of wardens and bot race winner of 2023-08-Shell contest. They found just one *Business Logic* issue. As we report in Table 54, wardens detect a *L3 - Missing Business Logic Checks* issue, titled *H01 - Lack of Balance Validation*. They report the fact that there is no balance validation inside two functions, `depositGivenInputAmount()` and `withdrawGivenOutputAmount()` of the `EvolvingProteus contract`. Shell developers have defined an invariant: *The pool's ratio of y to x must be within the interval [MIN_M, MAX_M)*. Without adequate validation, this invariant can be broken.

Category			C4 report	Mythril	Slither	Smartcheck
Business Logic	L3	Missing Business Logic Checks	1	0	0	0
		Overall	1	0	0	0

Table 55: Shell state-of-the-art automatic tools results.

Analysis tools results The results of automatic tools analysis are reported in Table 55. The only high issue was not found by them. Indeed, it is a *Business Logic* issue, and tools haven't any detector for such vulnerability. Furthermore, this issue is strongly related to a predefined invariant of the Shell protocol. Once again, the knowledge of smart contract context could help to detect vulnerabilities.

2.11 Stader

Stader				Derivative DApp	
	Contracts	Abstract contracts	Libraries	Interfaces	Overall
#	21	0	1	0	22
SLOCs	4191	143	0	0	4334
Pre-contest automatic tool analysis			Slither		
			Solhint		

Table 56: Stader's 2023-06 contest.

DApp description Stader is described in [27]. In Table 56 there is an overview of the 2023-06 contest. According to its documentation, "Stader is a non-custodial smart contract-based staking platform that helps you conveniently discover and access staking solutions. We are building key staking middleware infra for multiple PoS networks for retail crypto users, exchanges, and custodians".

Asymmetry (see Section 2.2) is a similar DApp: both allow stakers to deposit their assets and obtain a token, and this token grows in value according to staking rewards accruing.

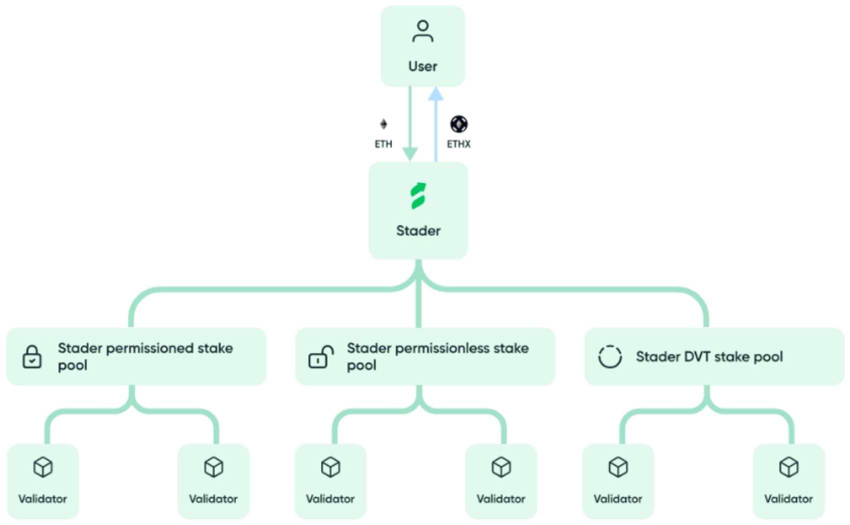

Fig. 4: Stader multi-pool architecture.

ETH staking in Stader is based on ETHx [28]. ETHx is a liquid staking token. Stakers can deposit their ETH and obtain ETHx tokens, that can be used to earn staking rewards and participate in other DeFi applications. The staked amount of ETH is used by nodes with validator purposes. The protocol has a multi-pool architecture, with permissioned and permissionless pools. The former is composed of trusted validators, while the latter can be accessed by everyone would operates nodes. In Figure 4 we report a picture of Stader architecture taken from the documentation.

Contest description This contest is mainly focused on multi-pool implementation. The **ETHx** contract is an ERC20 implementation. **PermissionedPool** and **PermissionlessPool** contracts are used in order to deposit ETH to **Ethereum deposit contract**. **StaderOracle** contract is the source of exchange rate and withdrawn validators for **PermissionedPool**. Finally, **StaderStakePoolsManager** allows users to staking ETH, minting ETHx, and managing staking rewards.

Category			Wardens		Bot Race	
			High	Medium	High	Medium
Business Logic	L0	Error in Business Logic Model	0	2	0	1
	L2	Erroneous Accounting Business Implementation	0	3	0	0
	L3	Missing Business Logic Checks	0	3	0	0
	L4	Wrong Implementation	0	2	0	0
Solidity	S5	Arbitrarily Code Injection	1	0	0	0
	S11	Relying on External Source	0	1	0	0
	S15	Erroneous `Pausable` implementation	0	1	0	0
Blockchain	B1	Frontrunning	0	2	0	0
		Overall	1	14	0	1

Table 57: Stader C4 Findings Overview.

C4 report description In Table 57 we report an overview of Stader findings of wardens and the bot race winner. As we can see, the bot race winner reports a *L0 - Error in Business Logic Model* finding. This seems strange: usually, automatic tools can't find *Business Logic* issues. This is not a vulnerability. The bot just detected the use of `owner` variable inside a modifier, that could represent a single point of failure. In Table 58 there is an overview of human high findings

C4 ID	C4 Title	Category
H-01	VaultProxy implementation can be initialized by anyone and self-destructed	S5 - Arbitrarily Code Injection

Table 58: Stader highs.

and corresponding category. Wardens just found 1 high issue, an instance of *S5 - Arbitrarily Code Injection*. In this case, the problem is due to the lack of initialization of the `VaultProxy contract` that allows an attacker to arbitrarily inject malicious code inside the vulnerable smart contract.

In Table 59 we report the wardens' medium findings and corresponding category. They found 10 *Business Logic* and 2 *Blockchain* issues. Moreover, they report 2 *Solidity issue*: a *S11 - Relying on External Source* and a *S15 - Erroneous `Pausable` implementation*. The former is due to a lack of checks on the behavior of an external source, Chainlink. The latter is a wrong implementation of the OpenZeppelin *Pausable* library (see Section 1.24).

In Table 60 there are Stader bot race winner findings and corresponding category. The bot winner reported just one medium finding: *M01 - The owner is a single point of failure and a centralization risk*. This issue created several discussions during the judgment phases of various bot races. It is not a vulnerability, and several wardens proposed to move this issue to *Analysis* category of Code4rena contests.

The bot winner reports these two findings in the `VaultProxy contract`:

```
function updateStaderConfig(address _staderConfig) external
    override onlyOwner {
```

```
function updateOwner(address _owner) external override
    onlyOwner {
```

In both findings, developers used the `onlyOnwer` modifier: it is also defined inside `VaultProxy contract`:

```
modifier onlyOwner() {
    if (msg.sender != owner) {
        revert CallerNotOwner();
    }
    _;
}
```

However, `owner` is a single address. Developers didn't provide a multi-role governance system. So, if the `owner` address is compromised, there is no way to recover the `VaultProxy` functionalities.

Analysis tools results We analyzed contracts from this C4 contest using Mythril, Slither, and Smartcheck. The results are reported in Table 61.

C4 ID	C4 Title	Category
M-01	Risk of losing admin access if 'updateAdmin set with same current admin address	L3 - Missing Business Logic Checks
M-02	pause/unpause functionalities not implemented in many pausable contracts	S15 - Erroneous `Pausable` implementation
M-03	Stader OPERATOR is a single point of failure	L0 - Error in Business Logic Model
M-04	updatePoolAddress functions always revert when updating existing poolId	L4 - Wrong Implementation
M-05	StaderOracle - Strict equal can cause no consensus if trusted nodes are removed before consensus	L2 - Erroneous Accounting Business Implementation
M-06	Protocol will not benefit from slashing mechanism when remaining penalty bigger than minThreshold	L2 - Erroneous Accounting Business Implementation
M-07	MEV bots can win all the auctions when Auction is paused	B1 - Frontrunning
M-08	Corruption of oracle data	L3 - Missing Business Logic Checks
M-09	depositETHOverTargetWeight() malicious modifications poolIdArrayIndexForExcessDeposit	L3 - Missing Business Logic Checks
M-10	Owner in VaultProxy.sol is address(0)	L4 - Wrong Implementation
M-11	ValidatorWithdrawalVault.distributeRewards can be called to make operator slashable	B1 - Frontrunning
M-12	ValidatorWithdrawalVault.settleFunds doesn't check amount that user has inside NodeELRewardVault to pay for penalty	L2 - Erroneous Accounting Business Implementation
M-13	No bidder has incentive to bid in the Auction except doing last-minute MEV due to fixed endBlock	L0 - Error in Business Logic Model
M-14	Chainlink's latestRoundData may return a stale or incorrect result	S11 - Relying on External Source

Table 59: Stader mediums.

C4 ID	C4 Title	Valid	Category
M-01	The owner is a single point of failure and a centralization risk	Y	L0 - Error in Business Logic Model

Table 60: Stader automatic findings.

Category			C4 report	Mythril	Slither	Smartcheck
Business Logic	L0	Error in Business Logic Model	3	0	0	0
	L2	Erroneous Accounting Business Implementation	3	0	0	0
	L3	Missing Business Logic Checks	3	0	0	0
	L4	Wrong Implementation	2	0	0	0
Solidity	S5	Arbitrarily Code Injection	1	1	1	1
	S11	Relying on External Source	1	0	0	0
	S15	Erroneous `Pausable` implementation	1	0	0	0
Blockchain	B1	Frontrunning	2	1	0	0
		Overall	16	2	1	1

Table 61: Stader state-of-the-art automatic tools results.

In this contest, automatic tools managed to find one *Solidity* and one *Blockchain* issues. In particular, all three found the *H01 - VaultProxy implementation can be initialized by anyone and self-destructed* high issue. Furthermore, Mythril managed to detect a *B1 - Frontrunning* issue, the *M07 - MEV bots can win all the auctions when Auction is paused* finding. We analyze deeper this in the next section.

3 Overview of the results

DApp	Contest Results		Tools Results		
	Human	Bots	Smartcheck	Slither	Mythril
Ajna	25	2	1	3	
Asymmetry	20		1	4	
Caviar	20				
EigenLayer	4			1	
ENS	7				
Frankencoin	21				
Juicebox	3				
Livepeer	5				
Llama	5	1			
Shell	1				
Stader	15	1	1	1	2
Overall	126	4	3	9	2

Table 62: Comparison between humans and bots findings versus tools findings.

References

[1] Georgia Weston. *Front-Running Attacks in Blockchain: A Complete Guide.* Dec. 2023. URL: `https://101blockchains.com/blockchain-front-running-attacks/`.

[2] Shayan Eskandari, Seyedehmahsa Moosavi, and Jeremy Clark. "SoK: Transparent Dishonesty: front-running attacks on Blockchain". In: *CoRR* abs/1902.05164 (2019). arXiv: `1902.05164`. URL: `http://arxiv.org/abs/1902.05164`.

[3] Wuqi Zhang et al. "Combatting Front-Running in Smart Contracts: Attack Mining, Benchmark Construction and Vulnerability Detector Evaluation". In: *IEEE Transactions on Software Engineering* (2023), pp. 1–17. ISSN: 2326-3881. DOI: `10.1109/tse.2023.3270117`. URL: `http://dx.doi.org/10.1109/TSE.2023.3270117`.

[4] Edvardas Mikalauskas. *$280 million stolen per month from crypto transactions.* Nov. 2023. URL: `https://cybernews.com/crypto/flash-boys-2-0-front-runners-draining-280-million-per-month-from-crypto-transactions/`.

[5] Kaihua Qin, Liyi Zhou, and Arthur Gervais. *Quantifying Blockchain Extractable Value: How dark is the forest?* 2021. arXiv: `2101.05511 [cs.CR]`.

[6] Consensys Team. *Denial of Service.* 2023. URL: `https://consensys.github.io/smart-contract-best-practices/attacks/denial-of-service/`.

[7] Igor Yalovoy. *Why Fomo3d 10,469 ETH Block Stuffing Attack Is Important.* Oct. 2019. URL: `https://ylv.io/why-fomo3d-block-stuffing-attack-is-important/`.

[8] T. Revoredo. *Blockchain consensus mechanisms and the role of game theory.* Nov. 2023. URL: `https://tatianarevoredo.medium.com/blockchain-consensus-mechanisms-and-the-role-of-game-theory-dd0985851f32`.

[9] Noama Samreen and Manar Alalfi. "Reentrancy Vulnerability Identification in Ethereum Smart Contracts". In: Feb. 2020, pp. 22–29. DOI: `10.1109/IWBOSE50093.2020.9050260`.

[10] Immunefi. *The Ultimate Guide To Reentrancy.* May 2023. URL: `https://medium.com/immunefi/the-ultimate-guide-to-reentrancy-19526f105ac`.

[11] Immunebytes. *Precision Loss Vulnerability in Solidity: A Deep Technical Dive.* Oct. 2023. URL: `https://www.immunebytes.com/blog/precision-loss-vulnerability-in-solidity-a-deep-technical-dive/`.

[12] Zuhaib Mohammed. *Unsafe Delegatecall (Part #2) — Hack Solidity #5.* Jan. 2022. URL: `https://coinsbench.com/unsafe-delegatecall-part-2-hack-solidity-5-94dd32a628c7`.

[13] Eszymi. *Ethernaut-Delegation — "dangerous" delegatecall.* Aug. 2022. URL: `https://medium.com/coinmonks/ethernaut-delegation-dangerous-delegatecall-e6d7a759d4f9`.

[14] A. Patel et al. "Ajna Whitepaper". In: *https://www.ajna.finance/* (2023). URL: `https://www.ajna.finance/pdf/Ajna_Protocol_Whitepaper_10-19-2023.pdf`.

[15] Asymmetry Finance. "Asymmetry Finance: Staked Ethereum Index Products with DeFi Applications". In: *https://www.asymmetry.finance/* (Apr. 2023). URL: `https://www.asymmetry.finance/whitepaper`.

[16] Caviar. *Caviar Docs*. Sept. 2023. URL: `https://docs.caviar.sh/`.

[17] EigenLayer Team. "EigenLayer: The Restaking Collective". In: *https://docs.eigenlayer.xyz/* (Nov. 2023). URL: `https://docs.eigenlayer.xyz/assets/files/EigenLayer_WhitePaper-88c47923ca0319870c611decd6e562ad.pdf`.

[18] ENS Team. *ENS Documentation*. 2021. URL: `https://docs.ens.domains/`.

[19] Frankencoin Team. *Frankencoin Documentation*. Nov. 2023. URL: `https://docs.frankencoin.com/`.

[20] Andrew Loo CFI Team. *What Happened to Terra?* 2023. URL: `https://corporatefinanceinstitute.com/resources/cryptocurrency/what-happened-to-terra/`.

[21] Juicebox Team. *Juicebox Documentation*. 2023. URL: `https://docs.juicebox.money/dev/`.

[22] Eric Tang Doug Petkanics. "Livepeer Whitepaper". In: *https://github.com/* (2017). URL: `https://github.com/livepeer/wiki/blob/master/WHITEPAPER.md`.

[23] Livepeer Team. *Livepeer Documentation*. 2023. URL: `https://docs.livepeer.org/developers/introduction`.

[24] Llama Team. *Llama Documentation*. Aug. 2023. URL: `https://docs.llama.xyz/`.

[25] Shell Team. "Shell Protocol White Paper". In: *https://github.com/* (Oct. 2020). URL: `https://github.com/cowri/shell-solidity-v1/blob/master/Shell_White_Paper_v1.0.pdf`.

[26] Shell Team. "The Ocean". In: *https://github.com/* (Feb. 2022). URL: `https://github.com/Shell-Protocol/Shell-Protocol/blob/main/Ocean_-_Shell_v2_Part_2.pdf`.

[27] Stader Team. *Stader Documentation*. 2023. URL: `https://www.staderlabs.com/docs-v1/intro`.

[28] Stader Team. "ETHx Litepaper". In: *https://www.staderlabs.com/* (Jan. 2023). URL: `https://www.staderlabs.com/docs/ETHx%20Litepaper.pdf`.